NASA Conference Publication 2114

Validation Methods for Fault-Tolerant Avionics and Control Systems -

Working Group Meeting I

Proceedings of a working group meeting
held at Langley Research Center
Hampton, Virginia
March 12-14, 1979

NASA

NASA Conference Publication 2114

Validation Methods for Fault-Tolerant Avionics and Control Systems -

Working Group Meeting I

ORI, Incorporated, *Compilers*

Proceedings of a working group meeting
held at Langley Research Center
Hampton, Virginia
March 12-14, 1979

NASA
National Aeronautics
and Space Administration

**Scientific and Technical
Information Branch**

1979

PREFACE

Fault-tolerant computer design is currently being directed toward a specified reliability requirement, namely, that the probability of failure to perform all vital functions during a mission will be no greater than 10^{-9}. The assertion that these computers meet this design requirement cannot be proven by any presently known procedure. The general problem is in fact incompletely defined, and one can only approximate such a proof by a complex process called "validation."

The first Working Group meeting provided a forum for the exchange of ideas. The state of the art in fault-tolerant computer validation was examined in order to provide a framework for future discussions concerning research issues for the validation of fault-tolerant avionics and flight control systems.

The activities of the Working Group were structured during the two-day session to focus the participants' attention on the development of positions concerning critical aspects of the validation process. These positions are presented in this report in the three sections entitled:

1. "The Validation Process For Fault-Tolerant Systems," by Albert Hopkins
2. "State of the Art in Validation," by Jack Stiffler
3. "Required Future Research," by Jack Goldberg

In order to arrive at these positions, the Working Group discussed the present state of the art in five sessions:

1. Fault-Tolerant Avionics Validation
2. Program (Software) Validation
3. Device Validation
4. Safety, Reliability, Economics, and Performance Analysis
5. Fault Models

Each session consisted of selected presentations followed by a free-wheeling discussion period. The session summaries and presentation abstracts are included in appendix A. Appendix B includes additional material submitted by two of the authors in Session II, and appendix C contains a list of the participants with their addresses.

The Working Group meeting was conceived and sponsored by personnel at NASA-Langley Research Center, in particular A. O. Lupton and Billy L. Dove.

TABLE OF CONTENTS

I. INTRODUCTION

Digital avionics and control systems show potential for undergoing a fundamental structural change with the introduction of reconfigurable fault-tolerant computers. These systems are planned for application to future CTOL transport aircraft used for commercial passenger carrying operations.

Redundant digital avionics and control systems are not new. Validation techniques for the current generation of digital avionics and controls have been developed and used on such systems as the F-111 redundant Mark II avionics, the B-1 redundant avionics, the F-18 quad redundant digital flight control, the F-8 triplex digital fly-by-wire system, and the Space Shuttle quad redundant avionics and controls with backup system.

However, with the possible exception of the Space Shuttle, these systems were not intended for passenger carrying operations. Consequently, proper pre-flight validation was important but not crucial to the lives of passengers. In the case of passenger-carrying operations, however, the avionics and controls failure probability must be of the same order as the basic structural failure probability. This extremely high mission reliability requirement imposes the need for the development of a reconfigurable, fault-tolerant avionics and control system for passenger aircraft, and of acceptable methods to demonstrate that the systems meet such rigorous reliability requirements. Loosely speaking, such a demonstration is termed "validation." The validation problem for such a system is considerably more difficult than for current redundant digital avionics and control systems. There are a number of factors which differentiate the fault-tolerant avionics and control system of the future from the multiple redundant avionics and control system of the present and recent past. These factors are discussed in the following sections.

- Transparency: The fault-tolerant avionics and control system is more complex than the multiple-channel redundant avionics and control system because of its inherent capability for both error detection and system reconfiguration. During the reconfiguration process the failed elements are removed from system mainstream operation and the remaining operable resources are redeployed where needed. This process occurs in a manner which is transparent to the user without necessary notification of failure. In other words, the system continues to operate without any discernible loss of function, performance, or safety. In a multiple-channel redundant system, each failure results in a significant reduction in failure margin and the pilot is notified of the failure so that contingency plans may be considered.

- Ultra-high reliability: The fault-tolerant avionics and control system has a much higher reliability than other redundant systems. The validation by traditional methods of high mission reliability on the order of 10^{-10} failures per hour for a 10-hour mission is practically impossible. Consequently, the validation process must use various new

tools and methods. The identification of this process was the purpose of this Working Group meeting which will be discussed in the following sections of this report.

- Totally integrated system: The fault-tolerant avionics and controls system integrates a variety of functions, such as flight control, propulsion control, navigation and guidance (outer loop control), flight management (navigation and flight data base management and system mode select), communications, ATC interface, system status and warning, in-flight and maintenance self-test, sensor conditioning, and control and display functions. This functional integration implies that the avionics and control fault-tolerant computer may be processing flight-critical, flight-crucial, and non-flight-critical tasks in the same computer at the same time. This possibility, however, does not forego architecture which may have some functional separation from one computer to the other. The major task facing prospective validators of such an integrated approach is proper classification of tasks under variable conditions. In addition, the fault-tolerant computer must be designed for generality of computation function because of the integration of modes cited above.

- Detectability of latent faults: The scope of the fault-tolerant computer and its capability for reconfiguration make it an inherently different kind of system from the multiple-channel redundant systems. The validation problem is compounded by the fact that latent failures may be disguised or hidden by the multiple-level recovery capability of fault-tolerant computers, or that latent faults may cause the fault class assumptions to be violated and thus cause unexpected errors.

It should be pointed out that "validation" is defined in its broadest sense in this report, and that validation for the fault-tolerant avionic and control system is a problem with many dimensions. In this report we do not mean to be limited to the narrow, formal validation process which occurs before system acceptance, but rather to encompass all steps leading to validation, including both formal and informal procedures used to record and substantiate the design process. Section II defines the validation process as used in the context of this Working Group report. The current practice of validation of redundant avionics was summed up by Don Dillehunt of Rockwell, who is involved in Space Shuttle software validation:

> "We do software validation by simulative testing on the Space Shuttle because we are familiar with it; we don't have time for other validation procedures. The problem with validation by testing is we really don't know when we're finished."

II. THE VALIDATION PROCESS FOR FAULT-TOLERANT SYSTEMS SUMMARY

2.0 INTRODUCTION

In order to make a rigorous case that a fault-tolerant system is a valid embodiment of its requirements, a systematic approach is required that is closely tied to the design process. At the same time, bridges between design, application, integration, certification, and deployment must be created. A variety of expertise is called upon to perform such tasks as software verification, hardware verification, analytic modeling, logical proof, simulation, test generation, fault injection, and failure modes and effects analysis. The expert "players" interact in a dynamic fashion. The results obtained from one can influence another's task. One of the objectives of the Working Group meeting was to establish a picture of the character of validation that could be used as a basis for a growing language for the exchange of ideas. The following is an attempt to capture the images which appeared to be tractable during the meeting.

Validation encompasses many activities over a substantial time period, using diverse implementations, and is conducted in various phases at various system levels, to various degrees of completeness, for various purposes. However obscure these parameters may seem, validation has a clear role, which is to confirm that a concept and its embodiments actually work well enough to justify a further commitment of resources.

This section begins with a thought model for the character of validation as it might be implemented in the future. This is not intended as a strict definition, but rather as a framework in which we can assemble the many detailed aspects that we shall be discussing in this or subsequent reports.

Section 2.1 treats the thought model in abstract terms. Section 2.2 presents more tangible examples and treats the significance of the larger context in which a single fault-tolerant computer operates. Some of the difficulties to be encountered are discussed in Section 2.3.

The final Section, 2.4, contains some observations made by members of the Working Group in the course of presentations and discussions on this subject.

2.1 THE CHARACTER OF VALIDATION

Rather than attempt to give a concise definition of the validation concept, we present a discussion of various aspects of validation, i.e., its character.

2.1.1 The Purpose of Validation

The purpose of validation of a fault-tolerant design is to generate and certify testable criteria of validity for repeated use at various times and places. Any copy of a design may be judged to be capable of meeting the design

goal if it meets all of these criteria (i.e., "passes" its tests). We may simply "ask" a system if it is "OK." Alternatively, we may subject it to a preflight end-to-end inspection.

2.1.2 The Process of Validation

The process of validation can be thought of as the creation and evolution of a body of abstract models. Tests of physical realizations and various forms of analysis are the evolutionary force. Figure 1 illustrates the concept. Results of tests and any other relevant experience, subjective or objective, are accumulated as they are obtained over time. Figure 2 gives examples of the various forms taken by models, realizations, analyses, and tests.

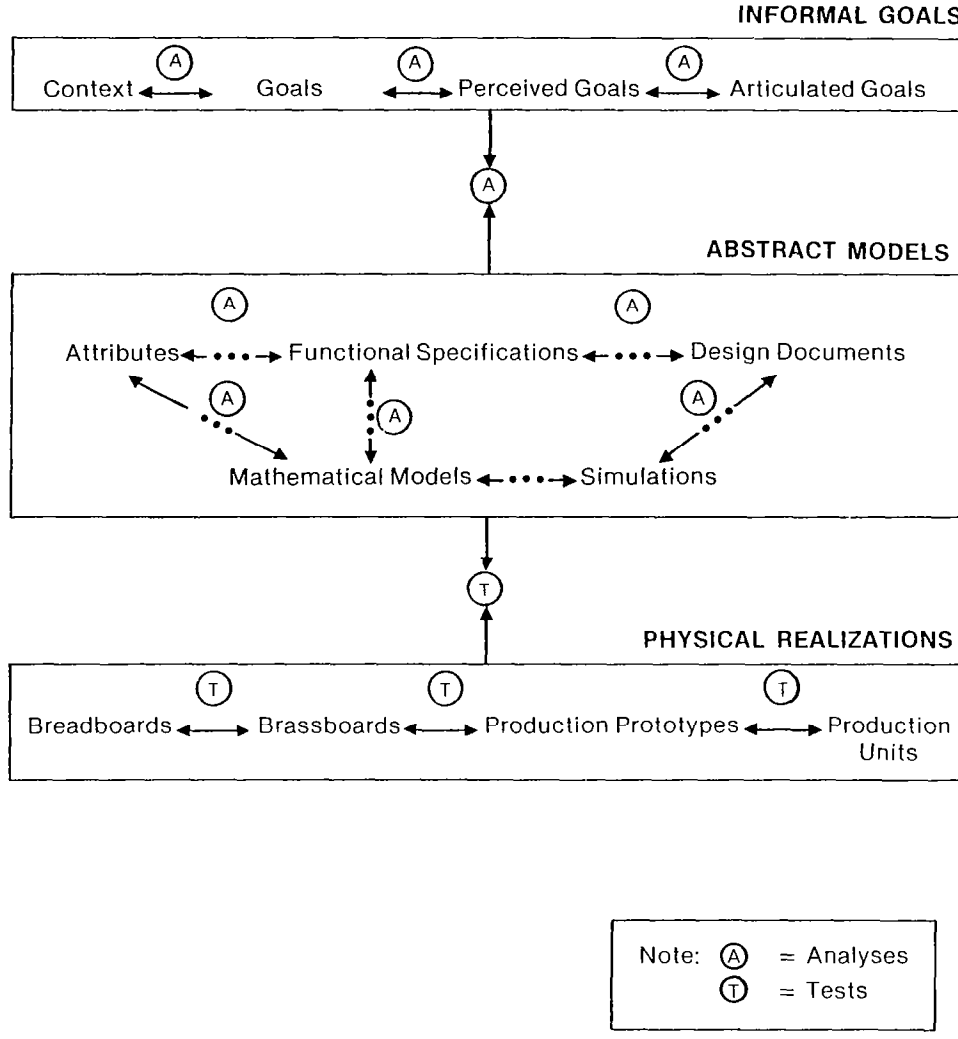


Figure 1.- The process of validation.

- ABSTRACT MODELS: GOALS, AGGREGATE (e.g., MARKOV) MODELS, FAULT TREES, BLOCK DIAGRAMS, FLOW DIAGRAMS, SCHEMATICS, SOURCE PROGRAMS, PROCESS AND MANUFACTURING DOCUMENTS, SPECIFICATIONS, REQUIREMENTS, SIMULATIONS, IDEAS, OPINIONS, ...

- PHYSICAL REALIZATIONS: COMPONENTS, CIRCUIT BOARDS, MODULES, BREADBOARDS, ENGINEERING PROTOTYPES, PRODUCTION PROTOTYPES, PRODUCTION UNITS, SUBSYSTEMS, SYSTEMS, ...

- ANALYSES: FMEA, FORMAL PROOF, STATIC VERIFICATION, CALCULATION, STATISTICAL ANALYSIS, CONJECTURES, SUSPICIONS, ...

- TESTS: COMPONENT, BOARD, AND MODULE TESTS, PROGRAM VERIFICATIONS, FAULT INJECTION TESTS, PERFORMANCE MEASUREMENT, ENVIRONMENTAL TESTS, INTEGRATION TESTS, FLIGHT TESTS, FLIGHT EXPERIENCE, ...

Figure 2.- Examples of validation tools.

The process integrates and coordinates experience as it is obtained from physical realizations (breadboards, prototypes, and production units), at any level (component level, module level, subsystem level, or system level), and is compared against abstract models, such as ideas, requirements, specifications, designs, simulations, analytic (e.g., Markov) models, and so forth.

2.1.3 The Measurement of Validation

The measurement of validation credibility, correctness, or assurance is elusive, much as is the measurement of a proof of Fermat's last theorem. In the final analysis, it is the judgment of an expert that matters. The validation must be measured from time to time in order to provide a credible basis for the acceptance of risk. The commitment to develop, to buy, to first fly, to produce, to deploy, and the act of certification all involve risk which can only be measured by means of the experience captured in the validation process. The measurement is a human function based on tradition, regulations, guidelines, training, experience, instincts, and so on. Thus there is to some extent a psychological component to the assessment of a validation. Therefore we may assume that a corresponding psychological component is apt to exist in the structure and contents of the validation file. A numerical measure would be desirable, and may be a legitimate goal of validation technology.

2.1.4 The Theory of Validation

A possible theory of validation is that the abstract models are evolved and corroborated with one another and with test data until these models are credible to a human user/inspector as vehicles for extrapolation of the data. According to this theory, the abstract models must be made to conform to one another, and to the physical realizations in the light of all test and analytic experience. Unexpected results, or "surprises," affect the process. The data obtained up to the time of the surprise may become irrelevant if the scope of the changes occasioned by the surprise cannot be contained. It is perhaps useful to think of the abstract models, which, although inexact, permit us to interpolate

among the discrete scattered data points obtainable through test. These abstract models form the basis for the generation of test vectors for the physical models. The results of the tests calibrate and confirm the abstract models. As the models become credible, the test domain can be broadened, which in turn allows yet greater credibility. We wish to extend the credibility of these models into areas not available to direct testing (e.g., the equivalent of 10^{-10} hours of flight-time and in-flight use). Stated another way, the results of tests are knitted together into an analytical structure from which one can infer the aggregate properties of subsystems and the system as a whole.

2.1.5 The Influence of Design on Validation

As long as the experience gained by testing remains in conformance with what the abstract models predict, we can suppose that we monotonically increase the degree of validation. A nonconforming test result, however, constitutes a "surprise," which may require a change in the abstract models or the physical realizations in order to produce conformance. The larger the change (in some undefined sense) the more experience is lost, and the greater the setback to validation.

A design that lends itself to validation will tend to compartmentalize all changes that need to be made to accommodate the surprises that occur during the validation process. This will minimize the experience leakage. It is unlikely that any significant system would ever be validated without surprises.

2.2 VALIDATION DYNAMICS

One view of validation dynamics is that it is a sequence of analytic and test-based confrontations among various models and realizations to see if they are compatible. This view (see fig. 3) shows on the right-hand side of the diagram various stages in the system life cycle which are connected by downward arrows to convey the sequential relationship among the stages of development. To the left are compound arrows which identify confrontations made during five distinct phases of a hypothetical aircraft design project that occurs over 8 to 10 years. These phases are roughly defined in figure 4. This confrontation view is somewhat primitive, in that it omits any explicit definitions of objectives of the various activities. Its virtue is that it shows a historical drift in the focus and intensity of the process that is not seen in the other views described here.

A second view of dynamics is the conceptual card file view, which emphasizes a sensitivity to surprises. In this view, the tests and analyses which are conducted throughout the process yield data which are filed and aggregated according to descriptors (i.e., categories) that have been anticipated. A surprise is a result for which no category exists. Accommodation of the surprise means enlarging, or otherwise altering, the descriptors list so that the new data can be filed in a way that maximizes the amount of old data that can legitimately be retained. In practice, this may or may not mean that a physical file is required, but it does imply a great degree of discipline in capturing every possible symptom that may be exhibited during the course of the validation process, and in making a sincere effort to trace its origin.

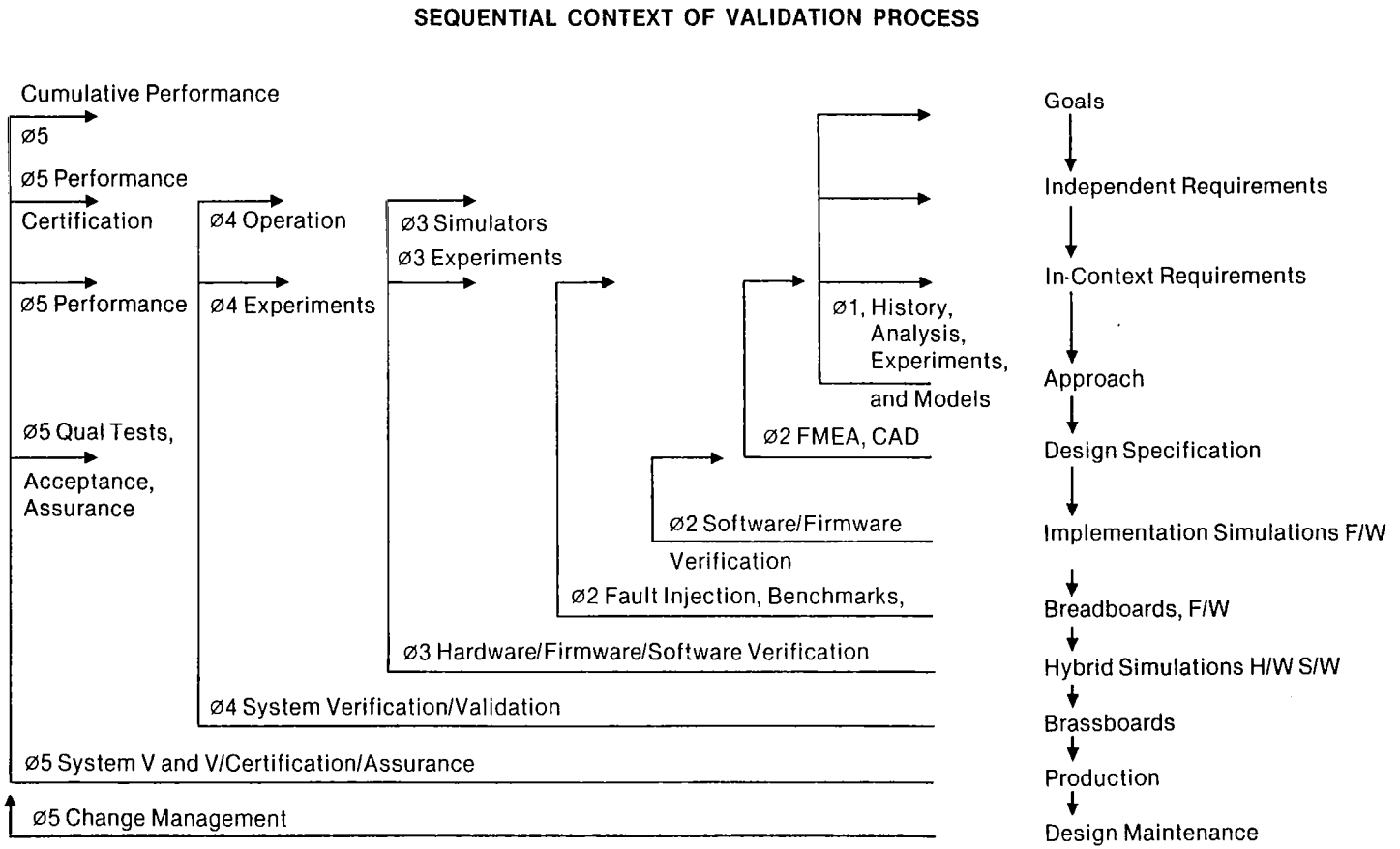


Figure 3.- Sequential context of validation process.

Phase	Activities	Tangibles
Phase 1	Historical Analogy Basic Models and Analyses Experiments	Emulations Simulations
Phase 2	FMEA, Injection, Basic Verification, CAD Analysis, Refined Models, Benchmarks, Estimates, Projections	Design Specs Schematics Flowcharts Listings Breadboards
Phase 3	Systematic Verification, Simulation, Testing	Software Assemblies Sensor/Effector Hardware Hybrid Simulations Iron Bird
Phase 4	Performance and Survival Verification Operational Experience Confidence and Consciousness	Aircraft Installation Brassboards Software Assemblies
Phase 5	Verification Qualification Certification	Prototype System Production Systems

Figure 4.- Validation phase summary.

A third view is what we have come in the course of our Working Group meeting to call the "ladder" diagram. It has its origin in the portrayal of the validation process as a step-by-step series of comparisons of increasingly detailed models, an example of which is shown in figure 5. In figure 5, the left-hand column shows a progression of models from the high-level informal problem statement down to the bitwise definition of the operational system. If the upward arrows can be replaced by proven implication, the net effect is a proven ability of the design to support the problem statement. The appeal of this view is its nearly unambiguous structure and economy of models. The relationship between the design and the realization is not covered by this ladder.

The basic ladder diagram approach is rigorous. It requires an exact design model, otherwise its implication is weak. A less rigorous diagrammatic form is shown in figure 6, where the models form a mesh, and where the intent expressed by the diagram is rigorous in some cases but flexible in others. Its two-way arrows are meant to imply that changes propagate through the models in both directions during the validation life cycle.

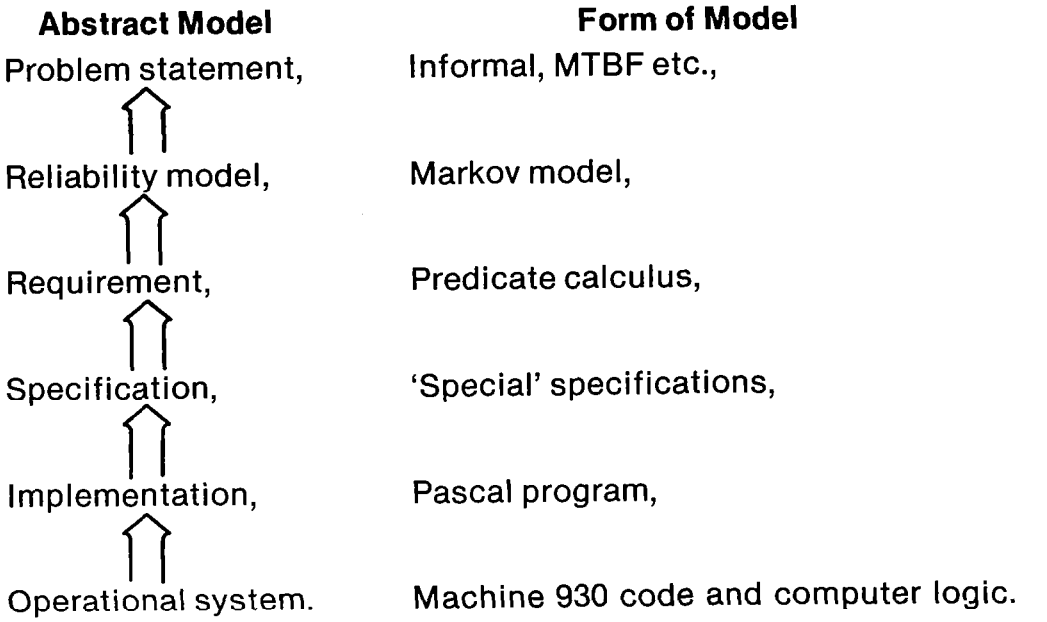


Figure 5.- Example of a ladder diagram.

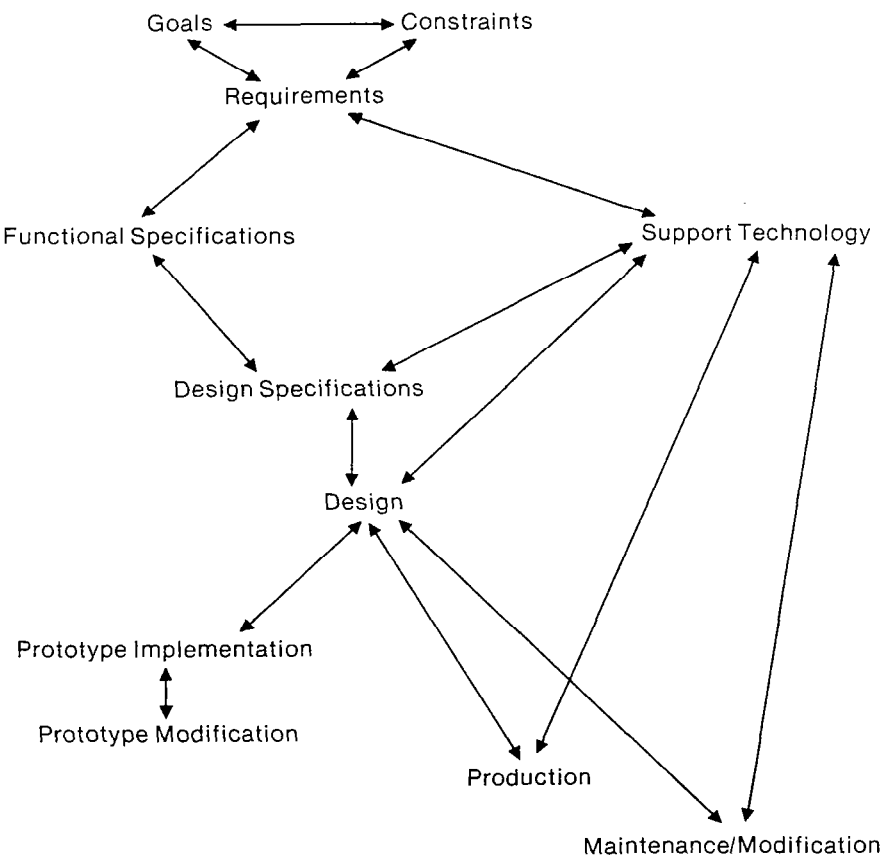


Figure 6.- Mesh form of ladder.

Figure 7 shows a third form of ladder which relies on incremental test data and analytic effort, and which has no explicit completion criterion. The double arrows in this case represent analytic inferences, and the single arrows represent tests. The form is very nearly the classical Failure Modes and Effects Analysis (FMEA) with verification by testing. When the user/inspector develops sufficient confidence in the performance, reliability, and economic models to justify his risk, the attributes that these models imply can be compared to the goals. It is intuitively felt that this form of ladder is potentially the least sensitive of the three to change owing to its informal structure.

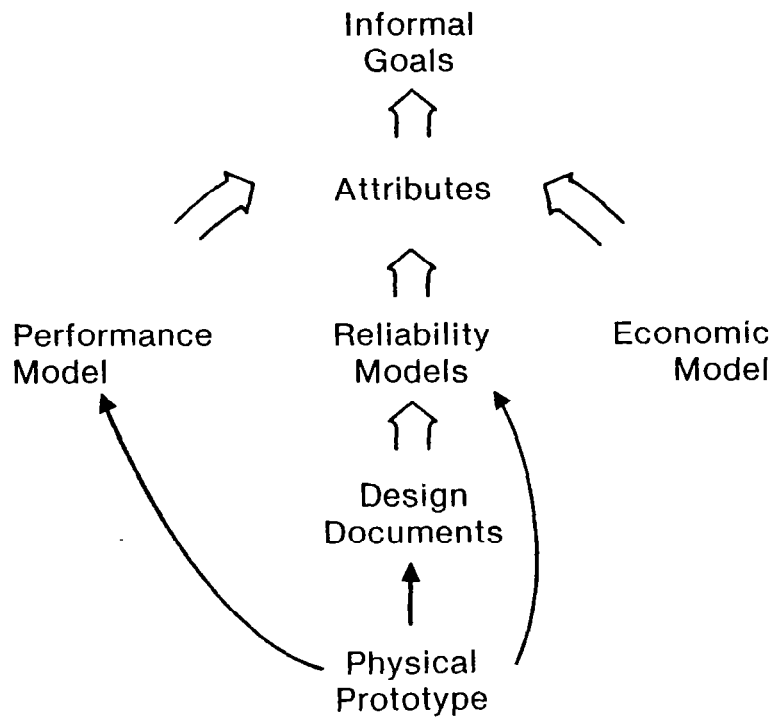


Figure 7.- Another ladder form.

2.3 MAJOR VALIDATION CHALLENGES

The thought model described in Section 2.1 is based on the premise that once the abstract models are correct, the proper testing will produce sufficient data to show that the risk is low enough to permit certification of the system. The premise cannot be taken for granted, however, for two major reasons. One is that the system being validated may simply not be reliable. The second is that even if it is reliable, it may be wholly impractical to implement the necessary variety of tests to impart the extremely high level of assurance or trust that is required to the human user/inspector. These two concerns are discussed below.

2.3.1 Unreliability

The aggregate reliability models, which indicate that a fault-tolerant system can exhibit a failure rate below 10^{-9} per hour, depend on certain assumptions about failure statistics and system behavior. Module failures are assumed to be independent and random, with constant hazard rate. The detection, diagnosis, and recovery actions are assumed to have an exponentially distributed completion time. There are known areas of uncertainty about these assumptions, however. The independence of faults depends on the effectiveness of common circuitry, as well as on the correctness of the software. The assumption of exponential distribution of recovery time is predicated on a notion of almost perfect detection and diagnosis capability. Pattern sensitivity, intermittency, and nonuniform signal perception are potential threats to the assumption. These are not easily dealt with or dismissed, and pose a major challenge to the skill of the designer and the evaluator alike. In some sense, this may be considered as an area of expected surprises. There is no certain way in which to verify that a design is immune from these threats, since exhaustive testing, even if possible, involves astronomical numbers of combinations.

2.3.2 Lack of Trust

A high value of predicted reliability obtained from an analytic model is not sufficient to satisfy a user/inspector that a validation is complete. Test data are required to lend credibility, or trust. The statistical concepts of confidence are widely used in component reliability engineering to quantify the trust that can be placed in a test. Confidence and reliability numbers compete with one another in the sense that a given experiment (set of data) supports the following two statements:

1. The device can be said to have a reliability of at least x with a confidence limit of y, or

2. To have a reliability of at least x+a with a confidence limit of y-b.

For purposes of validation, we are as much concerned with confidence (that is, how often our test might fail) as we are with predicted reliability (that is, how often our system might fail). Yet we believe that whereas we understand the relationship between confidence levels and reliability predictions at the component level, we do not even have a concept of confidence at the system level. If we did, then what level of confidence would be required for certification, assuming that the predicted reliability is satisfactory?

There is a notion, as yet unproven, that levels of assurance or trust, like reliability values, are amenable to improvement by redundancy as long as failure modes are independent. This is the basis of compartmentalized hulls and lifeboats for ocean vessels. Fault-tolerant systems may need to have layered recovery blocks for massive, though improbable, failures. If these layers of recovery are structured so as to possess proper coverage, then the trust that can be placed at the level of airplane safety may be substantially greater than the trust that it is possible to place on the nominal behavior of the computer or on any of the recovery blocks alone.

2.4 SOME OBSERVATIONS

The following are general observations which were not brought out earlier:

1. Validation is impeded by the following: a) the ambiguity in the definitions of requirements, b) uncertainties of data, c) unpredictability of faults and fault characteristics, d) faulty reasoning, e) the intrinsic complexity of fault-tolerant systems, and f) poor design. It is impeded also because digital systems lack continuity, inertia, and smoothness. Proof, informal analysis, simulation, and testing are strongly related and evolving technologies for validation. Design plays a key role in determining the feasibility of analysis for validation, and in dealing with the different levels of confidence that may be desired for system functions. The results of the validation procedure must be both rigorous and intuitively meaningful to the users of the system.

2. Fault-tolerant behavior should be an integral part of the definition of the performance requirements. The system can be viewed as processing faults as well as data. Faults may or may not be anticipated, and their consequences may or may not be acceptable.

3. Better models are needed for fault-tolerant systems that will allow formal treatment of fault types of histories, fault consequences, and fault tolerance mechanisms.

4. Validation must take account of the system hierarchy, from chip level to airplane level. It should also take account of other hierarchies, for example, levels of trust or assurance and levels of protection.

5. Simple structures and well-defined boundaries are fundamental urgencies in order for systems to lend themselves to validation.

III. STATE OF THE ART IN VALIDATION SUMMARY

3.0 INTRODUCTION

Numerous techniques have been developed and are currently being used for validation purposes. Although these techniques are generally inadequate for either the system complexities or the precision and confidence levels of interest here, they are the point of departure for any progress. It is therefore useful to examine the present state of these validation tools before attempting to identify areas in which further work is needed. For purposes of this examination, validation techniques are divided into five categories: testing, simulation and emulation, analysis, reliability modeling, and formal proof.

3.1 TESTING

One obvious method for validating an existing element (integrated circuit, software program, subsystem) is to test it and to compare its observed performance with its desired or specified performance. Indeed, testing is undoubtedly the single most commonly used tool for both hardware and software validation. Automated procedures are routinely used to test commercial and military electronic hardware, and environmental test laboratories are available to test hardware under a wide variety of operating conditions.

The major limitation of testing as a validation procedure is that the complexity of the element being tested may demand long, involved tests for complete checkout. Enormous numbers of tests may be required to validate the performance even of relatively simple logic circuits, particularly if those circuits contain memory. Standard tests for commercial LSI devices rarely result in greater than 95 percent coverage (with "coverage" here defined as the percentage of nodes in the circuit that actually change state during the course of the test) because the cost of higher coverage is prohibitive. Similar statements can be made with regard to conventional software testing procedures. It is therefore apparent that testing is not by itself an adequate method for validating an extremely high reliability system containing many interconnected LSI devices and software modules.

3.2 SIMULATION/EMULATION

Since there does not appear to be a universally accepted distinction between the terms "simulation" and "emulation," we will use simulation here to represent both concepts, i.e, to denote any procedure whereby one system is used to mimic the characteristics of another. Simulators are frequently thought of as computer programs. Such simulators have been designed to simulate the system of interest at levels varying from the gate level (the simulator generates signals representing the outputs of each of the gates of the simulated system) all the way to the functional level (the simulator duplicates functionally the simulated system). A number of simulation programming languages have been developed to facilitate development of computer programs for this purpose.

Simulation is a particularly useful validation tool during a design phase. It can be used to investigate the performance of a system before that system has been committed to hardware. It can also be used to measure a system's response to faults under various conditions, since faulty operation can be simulated as well as correct operation. This aspect of simulators is especially useful in the design of fault-tolerant systems.

Simulators are, and undoubtedly will continue to be, effective design tools. Since detailed simulators of fault-tolerant computers generally take longer (often much longer) to perform a given sequence of operations than the computer being simulated, simulation suffers from the same limitations as testing. It is not practical to simulate all situations of interest or even a large portion of these situations. Other tools are also needed to support the validation process.

3.3 ANALYSIS

Analysis in this context refers to those procedures whereby a system is subjected to a detailed examination with respect to its response to faults. Several systematic approaches have been used for this purpose. Failure Modes and Effects Analyses (FMEA) have long been used to ascertain the consequences of circuit malfunctions. Fault-tree and success-path analyses approach the same problem from the opposite direction. That is, rather than postulating a failure and then tracing its consequences, fault-tree analysis postulates a nonoperational situation and then attempts to identify those failures that could have caused that situation to occur. Similarly, success-path analyses attempt to identify all minimal subsets of equipment capable of performing a given operation successfully. (Such a procedure is obviously meaningful only when the system contains functional redundancy.)

Clearly these techniques are supplementary, with the most appropriate method depending upon the specific situation of concern. Equally clearly, such analyses become increasingly important as the complexity and redundancy of the system increase. Only by cataloging faults and responses, and in the process segregating them into equivalence classes, can we hope to reduce the inherent complexities of such systems to manageable level. Fortunately, this fault classification problem can be alleviated considerably by the fault detection mechanisms usually found in fault-tolerant systems: faults can often be classified in terms of the mechanisms by which they are detected.

Although analytical methods will continue to form an essential part of the validation process, some improvement in technique will be required. In particular, the high degree of certitude needed for avionics system validation necessitates consideration of many low-probability events which could be ignored in more conventional analyses. Because of this, and the complexity of the systems that will have to be analyzed, computer programs may have to be developed to aid such analyses if they are to continue to be effective validation tools in the future.

3.4 RELIABILITY MODELING

Reliability modeling is the process of representing the structure of the system of interest in a form amenable to probabilistic predictions. Such models must take into account the various failure mechanisms of the system (as determined, for example, through analysis and testing) and their internal relationships with each other and with other system functions. Classical probability analyses and various combinatorial and Markov techniques have been applied to this purpose with varying degrees of success. These techniques, while generally adequate for current systems, tend to get computationally cumbersome when faced with the large number of low-probability events that have to be accounted for in fault-tolerant avionics system validation.

The major limitation in reliability modeling, however, is in the inadequate statistical description of the faults that have to be modeled. Conventional hardware fault models ignore the fact that the hardware in question may contain design flaws, or that it may have been manufactured with inherent flaws which have not yet been discovered. Software design or implementation faults may also be present in programs of even moderate complexity. When such faults are uncovered, their effects on the system may well be as serious as those of any hardware fault. In any case, no reliability model can be considered complete unless it adequately accounts for such events. Recent efforts to develop statistical models to describe these events have met with only limited success.

More data are clearly needed with regard to both hardware and software faults so that good statistical fault models can be inferred. Such models must take into account not only fault types and rates of occurrence but their temporal properties as well; i.e., faults have to be modeled as stochastic processes rather than as random events.

3.5 FORMAL PROOF

Formal proof of system validity involves careful formulation of propositions concerning the desired performance of that system followed by a systematic demonstration that these propositions are in fact satisfied. Recent progress has been made in applying such techniques to proof of software program validity and in proving the validity of certain formal hardware specifications. The applicability of formal proof techniques may well be expanded in the future; it is not anticipated that such techniques will ever obviate the need for any of the other validation procedures described in the previous paragraphs, however.

3.6 CONCLUSION

All of the validation techniques currently in use will continue to be needed in the much more stringent future validation procedures. All of these techniques, however, have limitations that will have to be overcome if these validation goals are to be achieved. Some suggestions as to the directions these efforts should take are presented in the next section.

IV. REQUIRED FUTURE RESEARCH SUMMARY

4.0 INTRODUCTION

The preceding section discussed the state of the art in digital fault-tolerant system validation and some of its deficiencies. This section discusses research that is required to overcome these deficiencies. The goal of the research is to support the development of a more powerful technology for digital system validation that will be usable by industry and Government in validating future high-reliability, high-performance integrated avionic systems.

Some of the research is of a fundamental nature, and is aimed at problems that are common to all fault-tolerant systems. Other topics of research reflect particular characteristics of avionic systems. Many topics are currently being studied in various contexts, but the set of topics, taken together, is intended to comprise a unified program whose elements will be mutually supportive.

The research is organized into four groups, as follows:

- Fundamental research

  Research on the theory of analysis and design of fault-tolerant systems, aimed at providing a mathematical and physical foundation for building practical validation techniques

- Technology base

  Research on development of practical techniques and tools that are powerful and convenient for use by validation practitioners

- Data base

  Research on current experience and practice in digital systems, and on requirements for future fault-tolerant systems, aimed at assuring the relevance and realism of research on theory and techniques

- Case studies

  Experiments to help evaluate proposed validation methods.

The above four groups are definitely not intended to be pursued in isolation. Rather, research efforts should be organized that encompass, or at least coordinate closely, issues within all groups. These groups will be discussed in the following sections.

4.1 FUNDAMENTAL RESEARCH

The following are fundamental research goals:

1. Improve understanding of the validation process.

2. Improve understanding of the specification and design process.

3. Increase the effectiveness of formal and experimental verification.

4. Increase the power, computational convenience and fidelity of reliability and performance analysis.

4.1.1 Understanding the Validation Process

Models or paradigms are needed that will give comprehensive and consistent descriptions of the validation process. These models should allow clear expression of validation goals, techniques, and procedures and provide clear definition of general concepts, such as confidence, reliability, and validation.

Particular research problems are:

- Models for incremental validation processes that proceed through successive levels of the system life cycle, such as statements of user needs, statements of requirements, specifications, design, integration, implementation, operation, maintenance, and modification (cf. Section 2.2 on ladder diagrams). As a special case, models are needed for describing the validation of major changes in a system, such as redesign, or integration into new system environments.

- Clarification is needed for the roles of formal and informal verifications within the validation process.

4.1.2 Specification and Design

In order to achieve a high level of trust in the validation of complex fault-tolerant systems, it is necessary to have precise and understandable specifications for the functions that a given system is intended to perform. These specifications should have a high degree of mathematical rigor in order to allow formal verification of consistency between specifications and corresponding designs. Formal methods are needed that are appropriate to the specification of:

1. Hardware subsystems

2. Executive software

3. Avionic software

4. Avionic systems.

Different specification forms may be needed to express the particular characteristics of these items.

The feasibility of validation for complex fault-tolerant systems will also depend critically on the quality of the design. For example, feasibility of a Failure Modes and Effects Analysis (FMEA) will require careful design, so that large numbers of faults can be rendered equivalent. Elements of careful design include methods for structuring and partitioning system functions. Different methods may be appropriate for system elements such as hardware, executive software, application software, and the entire avionic system.

An important design approach is the use of fault tolerance in software and at the system level. Methods are needed to evaluate the effectiveness of this approach for structuring fault-tolerant systems.

4.1.3 Formal and Experimental Verification

Progress has been made in the theory and practice of formal verification (e.g., program proving). Much work is still needed to make this technique effective for avionic systems, including methods for treating numerical computations, asynchronous concurrent systems, and hardware. To be effective, this approach requires the use of hierarchical structure and formal specification.

Experimental verification, e.g., injection of faults in simulation models of fault-tolerant systems (both at the hardware and software levels) is in current practice. The method is useful for gaining insight, but it is costly, and it is difficult to assess the completeness of a set of experiments. Better theoretical understanding is needed to guide the construction of efficient test sets and to increase the power of the inferences about fault-tolerant performance that may be drawn from test results.

Better understanding is also needed on how to combine the methods of formal proof, simulation and physical testing so as to achieve the greatest level of confidence.

Future avionic systems will require the integration of independently timed and independently controlled subsystems. Faults in these systems may give rise to errors in synchronization, conflicts in control, and inconsistencies in data that are presently very difficult to analyze. New analysis techniques are needed to enable the verification of designs that attempt to prevent such errors.

4.1.4 Reliability and Performance Analysis

Progress has been made in increasing the realism of reliability models of fault-tolerant systems, but future systems will require even more complex models. Use of present methods will be very costly and of questionable accuracy. Present methods are based upon static (e.g., FMEA), analytical, stochastic, and simulation models. Research is needed that will increase the power, computational convenience, and fidelity in portraying dynamic behavior. Research is also needed on how best to combine these methods for use on high-reliability fault-tolerant systems. The following three aspects need more powerful analysis:

1. Occurrence of very rare events

2. Data-sensitive errors

3. Multiple faults.

4.2 TECHNOLOGY BASE

The following are technology development goals:

1. Development of methodologies and support tools

2. Development of presentation methods.

4.2.1 Methodology and Tools

To be effective in practice by system developers in industry and Government, the theoretical understanding derived from fundamental research studies needs to be applied systematically and with maximum automation. Research is needed to develop effective methodologies and support tools for a wide range of validation functions. These include at least the following four items:

1. Formal verification

2. Simulation

3. Physical testing

   --all the above for different system levels such as LSI devices, hardware, programs, computers, and avionic systems

4. Reliability and performance models of various kinds, including static, analytic, stochastic, and simulation.

In addition to methodologies for analysis, it would be beneficial to have available powerful design aids that can structure systems so as to make them easier to validate. Validation practice cannot assume full responsibility for the design function, but it should have sufficient impact on design practice to assure its own feasibility. Some research on how to build design tools that enhance validation is therefore justified.

4.2.2 Presentation Methods

Methods are needed to help in the presentation of the results of validation work in meaningful ways to users and purchasers of fault-tolerant systems. These may include notations and representations to summarize the results of analysis, and interactive systems that will aid users to explore the subject system and its validation.

4.3 DATA BASE

The following three types of data are needed:

1. Fault experience

2. Validation experience

3. Future needs.

4.3.1 Fault Experience

Data are needed on the frequency of occurrence of all fault types, including design faults in hardware and software, physical faults, and operational faults.

Data are needed about different fault forms, such as intermittent and correlated multiple faults, and about the effects of different environments.

New means are needed to collect such data, e.g., by incorporating fault monitors and recorders into operational equipment.

The value of a collection of tools for analysis and design would be greatly enhanced if they reflected a unified methodology for systems development. In the absence of complete control of the design process, tools may be needed to support a variety of design approaches. The goal of a unified methodology is nevertheless an important subject of research. Given such a methodology, the tools could comprise a powerful environment for the development of integrated avionic systems.

4.3.2 Validation Experience

Data are needed about the cost and effectiveness of current validation techniques, both established and experimental. These data will be important to the effective planning of large validation exercises.

4.3.3 Future Needs

Present reliability and performance goals for future avionic systems are very broad (e.g., 10^{-9} probability of failure for "advanced missions"). Data are needed for more precise estimates of future requirements for various avionic functions and subsystems.

4.4 CASE STUDIES

The following three types of useful case studies to aid in the evaluation of developing validation methods are needed:

1. Comparative evaluations

2. In-depth evaluations

3. Communication of results.

4.4.1 Comparative Evaluations

Case studies (e.g., of simplified flight-control systems) could be conducted to help compare competitive validation approaches, and to provide rapid evaluation for proposed modifications to an evolving approach. The approaches might include the full range of techniques, including specification, verification, and reliability modeling for various architectural approaches. The studies could also help to evaluate the reliability enhancement of hierarchical and fault-tolerant hardware and software structures and techniques for their validation.

4.4.2 In-depth Evaluation

Carefully chosen case studies could be used to evaluate the power of a proposed validation technique that has been developed to an advanced stage.

4.4.3 Communication of Results

At some stage in the development of advanced validation technology, some experiments would be valuable to help determine effective ways for communicating two kinds of results:

1. The presentation of validation results to users, purchasers, and certifiers

2. The transfer of validation technology to industrial system developers.

APPENDIX A - SESSION SUMMARIES AND PRESENTATION ABSTRACTS

OPENING REMARKS

Dick Smyth, Program Chairman

We welcome you to the first meeting of the Working Group on Validation Methods for Fault-Tolerant Avionics and Control Systems. As Billy Dove mentioned last night, NASA is currently planning new initiatives for the Aeronautics Avionics and Controls program beginning in 1981. These new-start avionics and controls programs include five broadly applicable technology areas:

1. Flight Path Management
2. Automatic Controls Technology
3. Crew Station Technology
4. Integration and Interfacing Technology
5. Fundamental Technology.

In addition the plan includes programs in five vehicle-specific technology areas:

1. CTOL Aircraft
2. V/STOL Aircraft
3. Rotorcraft
4. General Aviation Aircraft
5. Supersonic/High Performance Aircraft.

Prominent within both the broadly applicable technology areas and the vehicle-specific technology areas is the application of integrated, multifunction avionic systems including active controls technology and integrated flight control/propulsion control.

The thrust towards automatic flight controls within the active controls technology area is toward both crucial flight control modes and critical flight control modes, i.e., toward both:

1. Failure of flight controls which results in a vehicle catastrophe, and
2. Failure of flight controls which results in a reduction of the safe flight envelope.

This thrust to crucial flight control research in the 1980s will lead to operational critical flight controls with failure rate requirements of 10^{-9} per 10-hour mission in the 1990s. The only way we know how to achieve this high mission reliability with the current level of component reliability is by means of a reconfigurable, fault-tolerant avionics and control system.

The research on the next generation transport aircraft with active controls is underway. NASA is already sponsoring active control research using the following aircraft:

1. Boeing 747

2. Lockheed L1011

3. Douglas DC-10.

I had the opportunity to view a movie of the L1011 with 4-foot extensions on each wing tip to provide a greater aspect ratio for lower cruise drag and therefore better fuel economy (about a 3% increase). This aircraft needs active controls to provide damping of the wing bending modes. The movie clearly demonstrates that active controls really work.

The challenge of the working group is to define methods for validating the next generation fault-tolerant avionics and control systems which require the high reliability of 10^{-10} failures/hour. There is not enough time to test the system to validate the reliability, so the problem must be broken down into parts.

We have convened the most knowledgeable people in the country to address this validation problem, so let us begin.

SESSION I - FAULT-TOLERANT AVIONICS VALIDATION

Al Hopkins, Jack Goldberg, W. C. Carter, and John Wensley

SUMMARY

Session I addresses the validation process for fault-tolerant avionics and control systems. Four Working Group participants delivered the prepared comments summarized in the following sections. The principal points of these presentations and the general technical discussions which supported them are summarized below:

1. Validation is an ensemble of procedures involving objects (physical realizations and abstract models) and activities (physical tests and analyses) developed for the purpose of convincing users about the value of a system.

2. Analysis and testing are mutually supportive, e.g., analysis can guide testing, to make it more effective, and testing can substantiate the assumptions made in the analyses.

3. The abstract models that relate user needs to physical realization will typically comprise a hierarchy of abstract views. Validation of the abstract models requires the construction of a ladder of interconnected analyses.

4. A conceptual framework is needed for the discussion of the many complex issues that arise in the validation of systems in general, and fault-tolerant systems in particular.

5. Validation is impeded by the following characteristics:

- Ambiguity in the definition of requirements
- Uncertainties in the data
- Errors in the reasoning
- Intrinsic complexity of fault-tolerant systems
- Poor system design.

6. Validation of digital systems is much more difficult than validation of analog systems because digital systems lack the following:

- Continuity
- Linearity
- Smoothness.

7. Proof, informal analysis, simulation and testing are strongly related and evolving technologies for validation. Design plays a key role in determining the feasibility of analysis for validation, and in dealing with the different levels of confidence that may be desired for system functions. The results of the validation procedure must be both rigorous and intuitively meaningful to the users of the system.

8. Every system subject to validation is part of a hierarchy of systems, and every system has a life cycle of birth, use, changes and death in both design/evolution and physical realization, all of which need validation.

9. Fault-tolerant behavior should be an integral part of the definition of performance requirements. Faults may or may not be anticipated, and their consequences may or may not be acceptable.

10. Fault-tolerant systems need better models that will allow formal treatment of fault types and histories, fault consequences, and fault tolerance mechanisms.

11. The following additional observations and issues need further attention:

- Understanding of the dynamics of the validation process
- Relating poorly understood to well-understood system properties
- Introducing more powerful mathematical methods into all existing validation techniques.

Validation should eventually be described in many dimensions, including the following:

- Sequential Growth (see fig. I-1)
- Scope

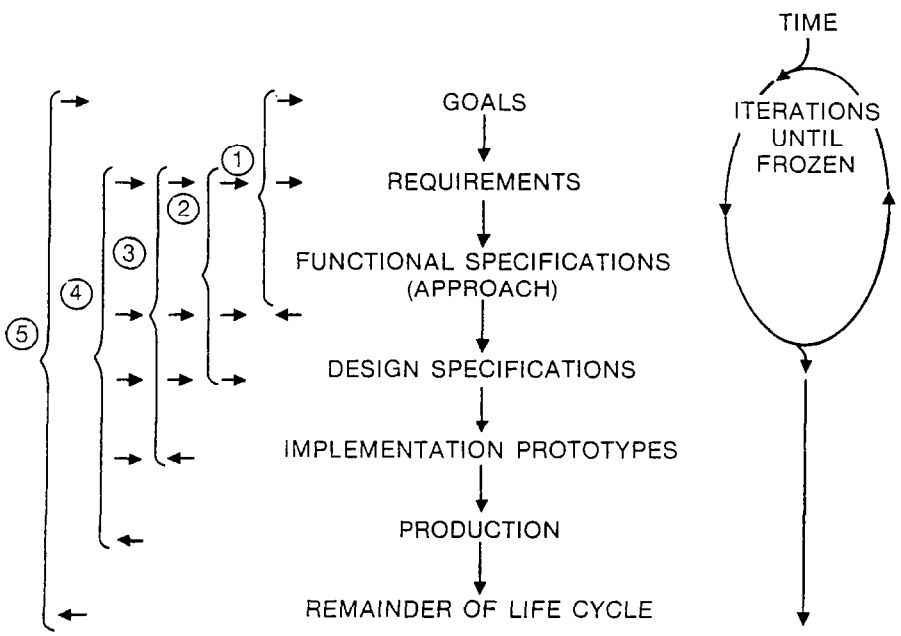
  Component ... Module ... System

- Degree

  Looks OK ... Probably OK ... I'm Convinced ... Certified ... Verified

- Depth

  One-time ... Repeated ... Self-validating.



① → (up to several years) abstract models, simulations.	Casual
② → (about a year) more detail, emulation, some proof.	Careful
③ → a (1-2 years) benchmarks, fault injection, more detail.	Systematic
→ b (up to several years) total system simulation and test.	Systematic
→ c (2-3 years) full certification, qualification, analysis.	Exhaustive
④ → (up to several years) acceptance tests, revalidation.	Methodical (by the book)
⑤ → (up to 20 years) logging, analysis.	Retrospective

Figure I-1.- Validation sequence model.

GENERAL VALIDATION ISSUES

Al Hopkins

OBJECTIVES

Our principal objective is to identify (if we see them), or to develop (if we can), methodologies for validation that are applicable over the domain that includes all of the viable candidates for critical systems. Validation can be formally defined in various ways, which is not a good state of affairs if we are trying to study it systematically. Rather than try to formally define it, let me offer an informal definition (see fig. I-2); then let us try to describe it, in hopes that we can agree on basic principles.

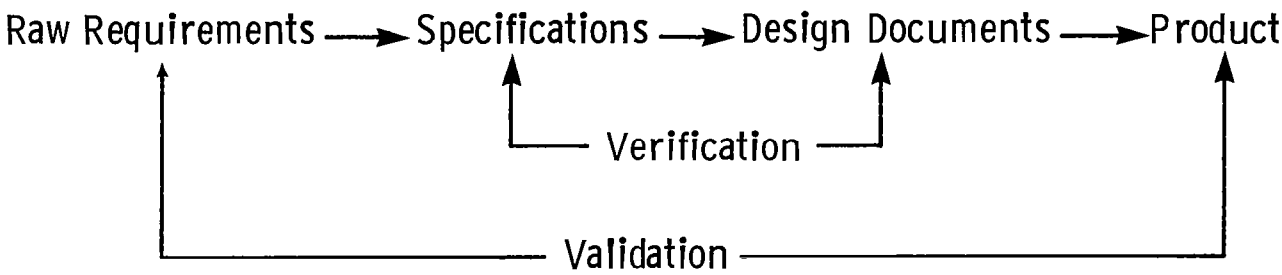


Figure I-2.- Informal definitions of validation and verification.

PURPOSE OF VALIDATION

The purpose of validation is to provide expeditious criteria for judging whether a member of a group of systems meets its requirements. This purpose may be summarized:

- To decide if a given system in the field is "OK"
- To develop confidence in a given system prior to use in the field
- To convince others of the adequacy of a given system to perform its intended function in the field.

To achieve these goals, we conduct a "system validation." If the system passes, and if the validation procedure possesses the proper credentials, we declare the system "OK" until further notice within the limits of the validation assumptions. This process is possible only if the validation vector, the credentials, and the system design have been validated as an ensemble. The criteria of validation may be self-test or end-to-end system test in conjunction with a historical data base.

VALIDATION PROCESS

The achievement of these criteria is presently an iterative process of matching physical models with abstract models, and eventually using the latter as a means of extrapolating data taken from the former. Both types of models evolve during the process. The validation process can be characterized as follows:

- Statements of requirements, abstract models, and physical models are employed throughout a dynamic process of design and assessment
- Abstract models are the means by which requirements are processed to become physical models, and are the medium in which physical models are confronted with requirements
- Analysis relates one abstract model to another or to a requirement. Test relates one physical model to another or to an abstract model.
- Testing is conducted to build credibility in an abstract model or to indicate its nonconformance. When a change is made to restore conformance, the prior credibility may or may not be voided.

Examples of this process include:

- Abstract models: aggregate (e.g., Markov) models, block diagrams, flow diagrams, schematics, source programs, process and manufacturing documents, specifications, simulations, ideas, opinions...
- Physical models: components, circuit boards, modules, breadboards, engineering prototypes, production prototypes, production units, subsystems, systems...
- Analyses: FMEA, formal proof, static verification, calculation, statistical analysis, conjectures, suspicions...
- Tests: component, board, and module tests, program verifications, fault injection tests, performance measurement, environmental tests, integration tests, flight tests, flight experience... .

The process of validation is complete when a human assessment so declares it. It should not be surprising if we find that validation processes are often structured to reflect this fact.

Validation can be conducted at the component level, the system level, or any level in between. The process and the means of validating a system usually depend on the consequential cost of an erroneous result.

THE STRUCTURE OF VALIDATION

Figure I-3 presents a way to view the validation structure. This structure includes vast ensembles of requirement statements, abstract models (ideas, aggregate models, virtual system models, design documents, process documents, simulations, etc.), and physical models (breadboards, prototypes, and production units) which are treated today by numerous tests, analyses, and observations in what appears to be a logical patchwork.

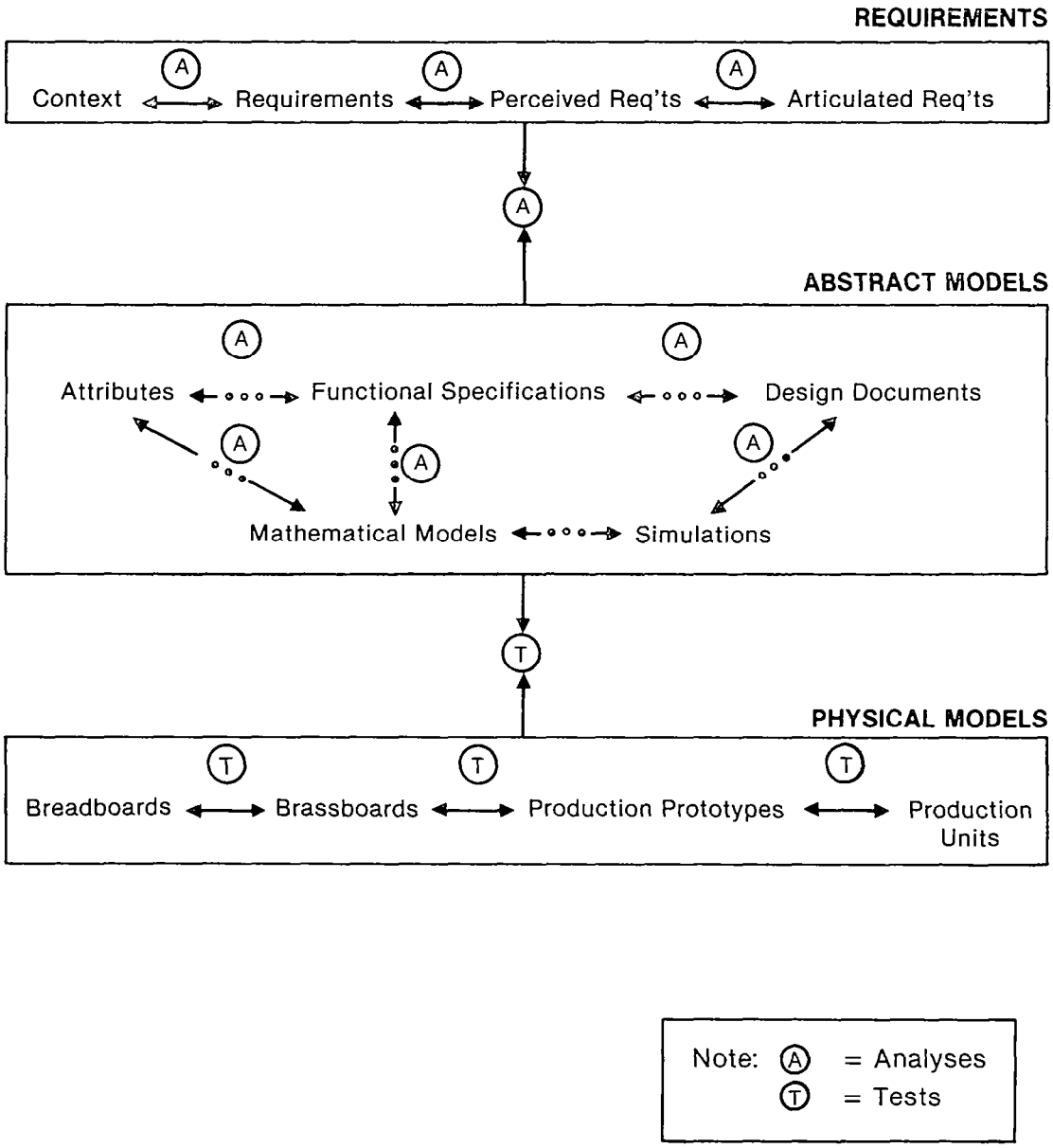


Figure I-3.- A validation structure.

The current status of this structure is characterized by the following three items:

1. An ensemble of abstract models evolves through test and analysis. If the design and the models are sufficiently robust and adaptable, confidence will be developed despite occasional setbacks.

2. The ensemble of abstract models becomes increasingly depended upon to interpolate and extrapolate the scattered data points obtained from tests on physical models.

3. The objective of validation is achieved when the responsible human agent develops sufficient confidence in the abstract models that extrapolated test data provide a satisfactory "answer."

This structure is severely limited in the following three areas:

1. Fault models are imprecise. We are limited by how precisely we can predict probabilities of faults or even fault classes.

2. Testing is incomplete. We are limited by not knowing what inferences can be made about the presence or absence of faults in a system on the basis of a test or a series of tests.

3. Confidence is intangible. We are limited by having to rely on a subjective appraisal of the credibility of a validation process.

It is possible that architectures can be devised which will lessen the dependence on such a fragile methodology, and the following items are offered as reasons for (at least guarded) optimism:

1. Architectures can go far toward pushing malfunctions into equivalence classes

2. An unbounded fault detection time does not necessarily imply an unbounded system failure rate

3. Recovery options exist even if a system anomaly occurs (cf. performability)

4. Engineering models are under construction that will serve as experimental testbeds for redundancy management strategies

5. These same engineering models will help to focus validation methods and theories.

GENERAL CONCEPTS OF VALIDATION

Jack Goldberg

INTRODUCTION

These remarks are intended to provide a preliminary framework for discussing the validation of fault-tolerant systems. We shall consider the following three items:

1. General issues in system validation
2. Special issues in the validation of fault-tolerant systems
3. Elements of a framework for the validation process.

GENERAL ISSUES IN SYSTEM VALIDATION

We consider the familiar view that system validation is a process by which a seller attempts to convince a buyer or user (and possibly an indemnifier) that a product will meet the buyer's needs. The following five difficult issues arise from this simple view:

- Buyer's Needs. Validation is meaningful only with respect to a statement of needs or requirements. Unfortunately, most requirements statements are incomplete and ambiguous. One must therefore expect that a full requirements statement will have to evolve from the validation process itself. Note that a requirements statement may include demands as to what a system must not do, as well as what it must do.

- Buyer's Confidence. An honest seller should admit that some features of his complex system are physically or economically unknowable and unpredictable, that the system environment cannot be fully controlled, and that not all system changes can be predicted.

  The prudent buyer must recognize that lengthy or intricate arguments will contain errors in reasoning, and that demonstrations are necessarily limited. This awareness usually leads to the deliberate choice of numerous redundant validation approaches, chosen to reduce the number of common errors. Rigorous logic, based on clear assumptions, can save redundancy in the validation process.

- Layers of System Contexts. Every product to be validated will be a subsystem within some context, and that context will itself be a subsystem within some higher context. For example, computer hardware, operating system, flight programs, avionic system, aircraft, and fleet form layers of progressively complex systems. Validation must be performed for the evolving product as applied in its particular context.

- Validation and the System Life Cycle. If a system is to be validated at all, it needs to be validated every time it is changed. Figure I-4 is meant to show that this concept applies clearly to all the stages of initial production, correction of design, upgrading, repair, and extension.

  - The Need for Revalidation After Change
  - The Benefit of Incremental Validation

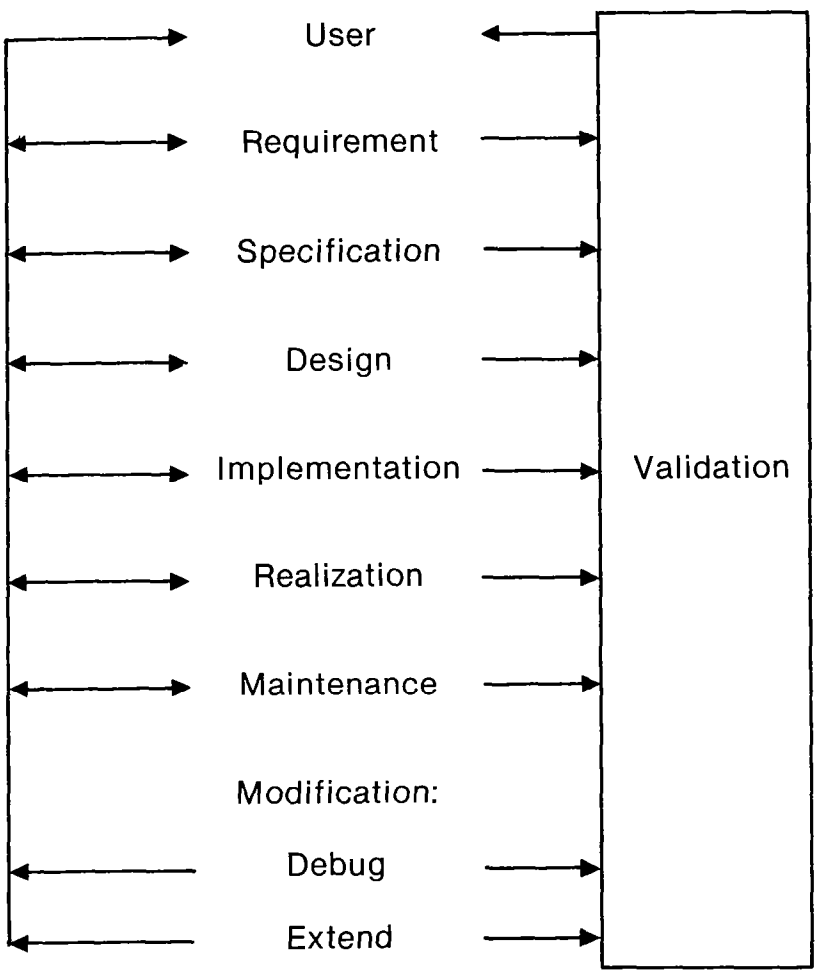


Figure I-4.- Validation and the system life cycle.

Perhaps not so obviously, validation is also valuable when applied to the numerous stages in the evolution of a design. Given proper methods of abstraction, it is possible and may even be essential to validate that a design is consistent with its requirements, even before it has been implemented. Such evolutionary stages include requirements, specifications, design, implementation, realization (production) and application.

Each successive state may be examined meaningfully for consistency with respect to the previous one.

- Approaches to Validation. Proof, analysis, simulation and testing are all methods which form a continuum of validation methods. Each method has particular advantages and limitations (summarized in fig. I-5) which determine when and how it should be employed.

Approach	Benefit	Limitation
• Proof	Complete coverage of input domain	• Need for special skill • Uncertainty in assumption
• Informal Analysis	Fast discovery of design blunders	Mental lapses
• Simulation	Low-priced realism	Poor coverage of input domain
• Testing	Checks physical assumptions	Extremely poor coverage of inputs

Figure I-5.- Approaches to validation: proof, informal analysis, simulation, and testing.

SPECIFIC ISSUES IN FAULT-TOLERANT SYSTEMS VALIDATION

We take the view that fault-tolerant systems validation is a special case of general system validation. The following four special problems are noteworthy:

- Fault Tolerance as an Integral Part of Requirements. Fault tolerance cannot be separated from performance and other requirements. That is, a requirements statement should specify how a system's performance is affected by a sequence of faults, e.g., in reduced functionality, speed, precision, etc. Validation must, therefore, deal not only with two system states, e.g., perfect and failed, but with a continuum of intermediate "partially failed" states.

- The Need for Models of Fault-Tolerant Systems. A validation methodology for fault-tolerant systems will require a general model for such systems in order to overcome the vagueness and inconsistency of present terminology and to assure consistency and completeness of analysis. Such models do not exist. Two possible approaches are:

- Consider the fault-tolerant system as a fault processor. (See fig. I-6.) In this view, a fault-tolerant system is one that processes two types of "data": (a) numerical and symbolic data, to be treated as inputs to programs, and (b) "faults," i.e., perturbations to idealized functions, to be treated as "inputs" to a fault tolerance process (e.g., detection/reconfiguration).

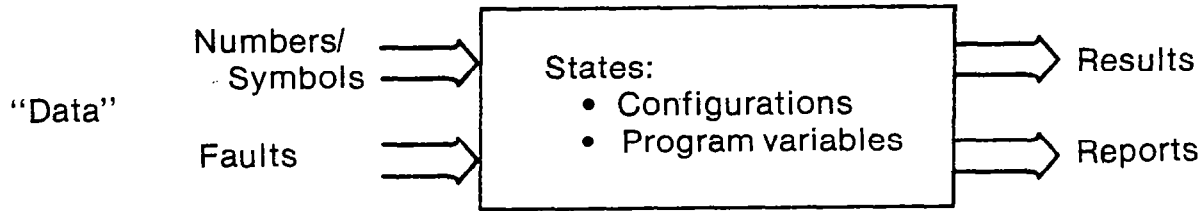


Figure I-6.- Fault-tolerant system as a fault processor.

- Treat the fault-tolerant system as a family of systems (fig. I-7). In this view, there is an ideal, fault-free system at the center of a function space surrounded by a set of systems whose input-output processing functions are differentiated by unique fault sets.

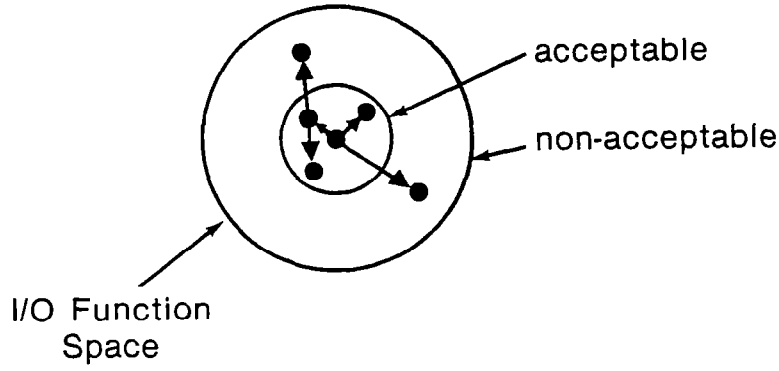


Figure I-7.- Fault-tolerant system as a family of systems.

Such views need to be refined so as to enable description of fault histories, fault types (design, implementation, operation), fault consequences, and fault tolerance procedures.

- Faults and Consequences. Faults are events that are undesirable and, in general, unpredictable. Their consequences may be unacceptable, or acceptable with various cost penalties. Faults may be either anticipated or unanticipated by the designer. All four combinations of acceptability and anticipation (represented in fig. I-8) are meaningful.

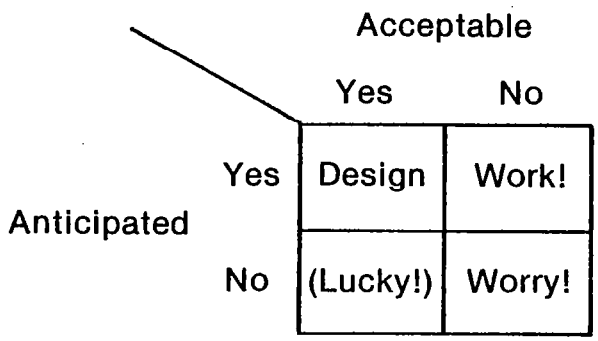


Figure I-8.- Taxonomy of faults and consequences.

- The Role of Design. System validation is difficult in general, and is particularly difficult for fault-tolerant systems, due to the infinite variety of behavior that may be induced by likely fault histories. Validation analysis for fault-tolerant systems will succeed only if the systems are designed to be very simple and to have strong and very well defined interior and exterior boundaries.

  Since validation is a costly and error-prone process, it would be prudent to design systems so that their most critical functions can be analyzed with the highest degree of confidence. This principle (sketched in fig. I-9) could be extended to yield a system that is structured into rings of differing confidence by criticality levels.

A FRAMEWORK FOR FAULT-TOLERANT SYSTEM VALIDATION

Based upon the preceding discussion, it appears that any framework for fault-tolerant system validation must include the following four elements:

1. Formal statement of requirements for behavior over specified fault classes.

2. Design philosophy for:

   - levels of protection
   - levels of confidence of validation
   - simplicity of analysis.

3. Incremental validation over all stages of design and use.

4. Integration of proof, informal analysis, simulation and testing.

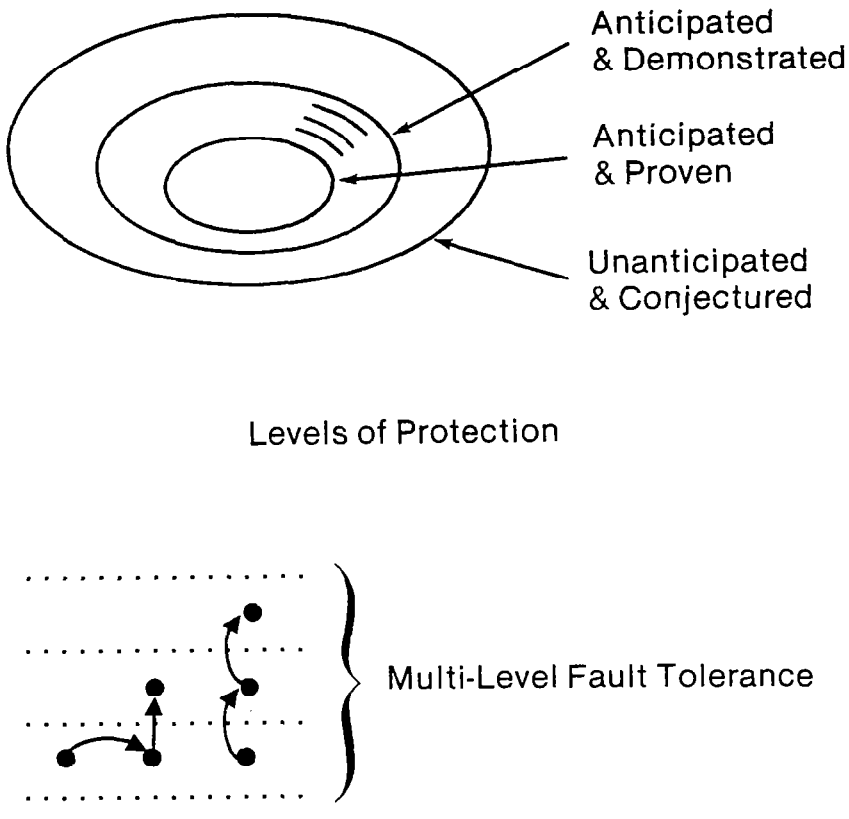


Figure I-9.- Levels of confidence of validation and levels of protection.

OVERALL CONCLUSIONS

The following six items are offered as overall conclusions from the preceding discussion:

1. A conceptual framework is needed for the discussion of the many complex issues that arise in the validation of systems in general and fault-tolerant systems in particular.

2. Validation is limited by the ambiguity of requirements/definitions, by uncertainties of data, by unpredictability of faults and fault characteristics, by faulty reasoning, by the intrinsic complexity of tolerant systems, and by poor design.

3. Every system subject to validation is part of a hierarchy of systems, and every system has a life cycle of changes in both design/evolution

and physical realization, each of which needs validation. Proof, informal analysis, simulation, and testing are strongly related and evolving technologies for validation.

4. Fault-tolerant behavior should be an integral part of a performance/requirement definition. Faults may differ in their acceptability to users and in the extent of their anticipation by designers.

5. A general model is needed for fault-tolerant systems that will allow formal treatment of fault types and histories, fault consequences, and fault tolerance mechanisms.

6. Design plays a key role in determining the feasibility of analysis for validation, and in dealing with different levels of confidence that may be desired for different system functions.

VALIDATION AS A SYSTEM DESIGN CONCEPT

W. C. Carter

VALIDATION DEFINED

By definition, a dependable computing system attains its expected system behavior regardless of the appearance of faults. Showing that a computing system, as designed, will meet its dependability goals is called validation. Webster defines validating an assertion as:

1. "Showing that the assertion is founded on truth or fact," or

2. "Showing that the assertion is capable of being justified or defended."

Thus, validation is a social process; validation is a complex process, and depends upon the design techniques used.

A typical system life cycle will be summarized in figure I-10. Initially, a set of goals is determined, and the appropriate constraints are usually evident. Using innovative ideas and experience, the system designers develop a set of requirements which satisfy the goals as modified by the constraints. The informal requirements become the formal function specifications by further refinement and forecasting, and the necessary accompanying support technology requirements are developed. The functional specifications which are application oriented are combined with system control to become design specifications from which a design is developed. To aid in determining the validity of the design, a prototype is implemented and modified (causing design modifications) until a product is defined. The product is produced from the modified design, using the facilities of the support technology. The product is maintained in the field, and further modifications are made to improve performance, save money, or both.

Functional Specifications

Note that the requirements show the gap between the goals and reality. The validation goal is to show, by analysis using simplified models, that this gap is not too large. Primarily because this process tends to be technically informal, this step is too often skipped.

The requirements, in English, are now modified to produce the functional specifications. These specifications are informal in structure, unambiguous, and define the necessary tasks which must be performed by the dependable system. The validation assertion is that the tasks in the functional specifications can be combined to satisfy the requirements. The validation techniques use more detailed mathematics, usually including some simulation. The tasks are ranked according to their criticality and probability of successful implementation. The strengths and weaknesses of the functional specifications in meeting the

requirements are analyzed, ranked, and recorded. The validation output is a report on the adequacy of the functional specifications and an estimation of the probability of successful project conclusion.

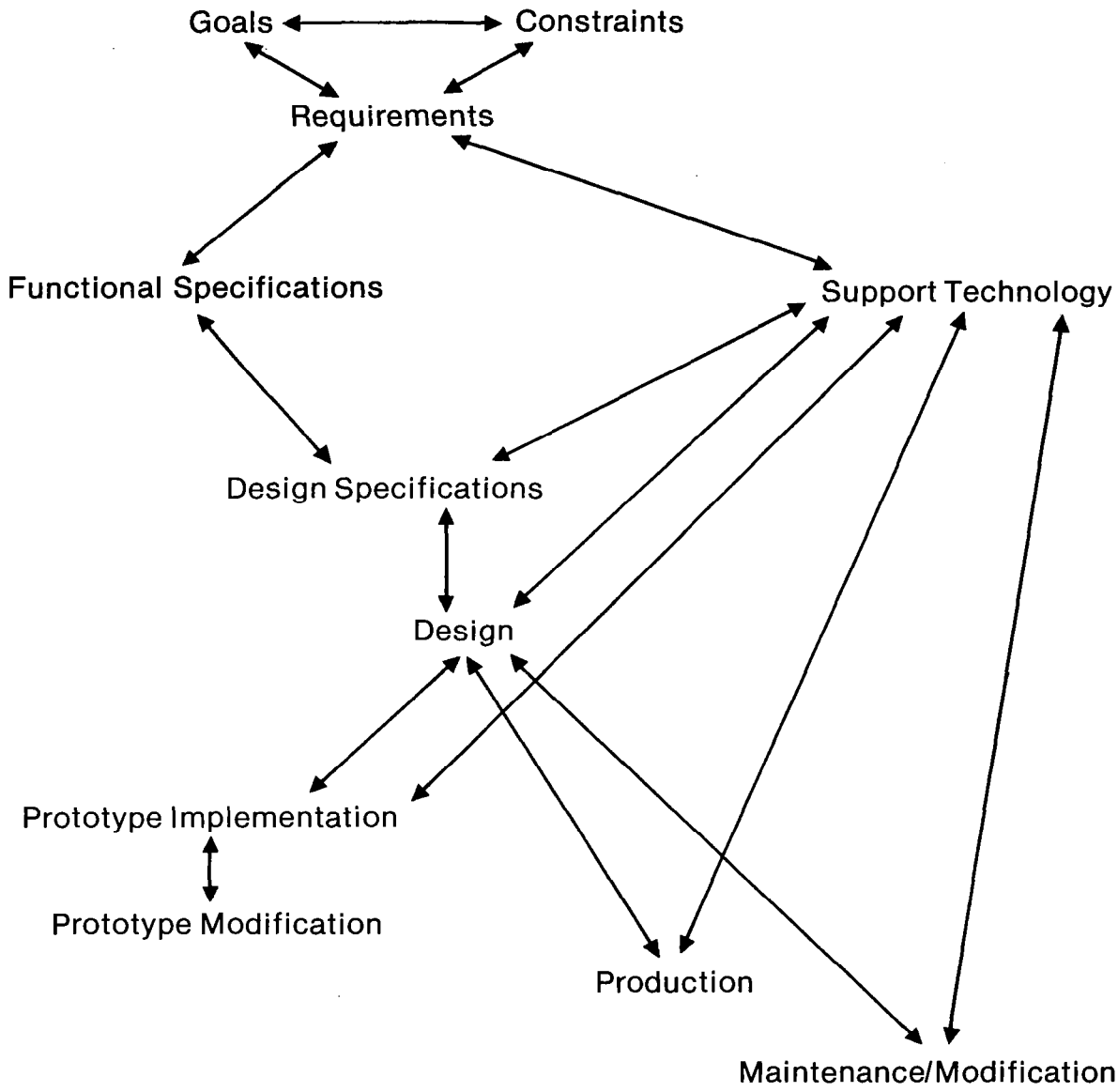


Figure I-10.- Representation of the system life cycle.

Dependable Computing

For dependable computing, the central issue is recovery. The errors caused by faults must be detected and their effect overcome. There are two main techniques (see fig. I-11) for error detection--structural and functional.

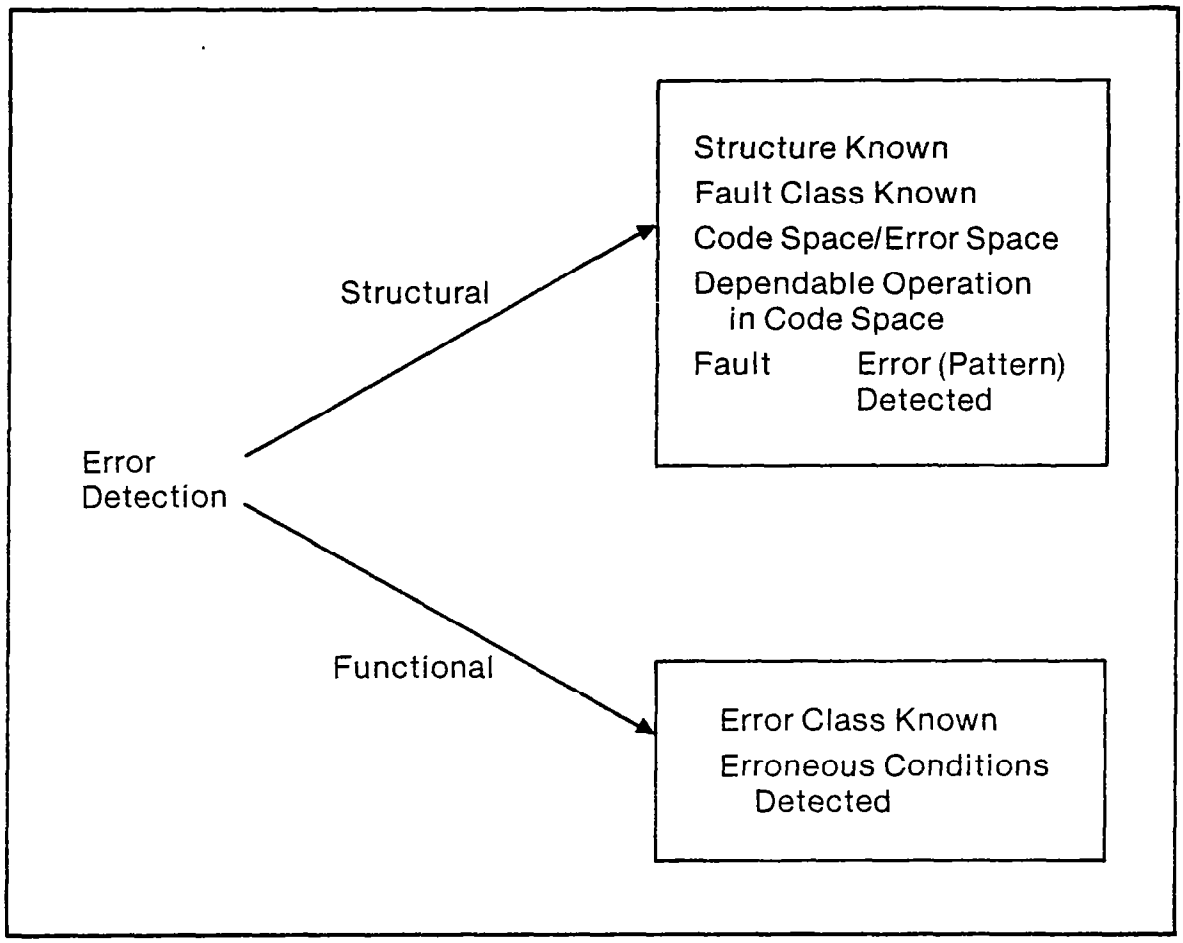


Figure I-11.- Techniques to detect and overcome effect of errors.

For structural detection, techniques are analyzable. A wide (and sufficient) class of fault modes is assumed for the system. Now a set of codes is devised which divides the system structure into code space and error space elements. Dependable operation occurs in code space. A fault in the fault class produces an error which changes an element in code space into an element in error space, so the error is detected. In functional detection techniques a wide class of erroneous system operating conditions is defined, and methods are devised to detect such conditions.

Testing for errors may be based on either structural or functional techniques. For a program, structural testing implies testing using the program's graph structure. For example, execute all branches, execute all branches both ways, etc. For functional testing, a program is run using selected data as input, and the correct operation of the program is monitored. For hardware, structural testing uses test patterns that show the presence (or absence) of component faults. Functional testing ensures that input data are correctly processed according to the system specifications. Structural error checking uses coding techniques; functional checking uses properties such as memory bounds or capability checks.

Design Specifications

Functional specifications, the set of necessary tasks, are transformed into design specifications. Control is specified so that the set of necessary tasks is available as a subset of a single total system. The validation assertions which must be satisfied are: 1) Each task is performed correctly and 2) The combination of tasks satisfies the functional specifications. The design techniques used to go from functional to design specifications use many ideas. In a few cases a formal hierarchical structure with inviolate interfaces is used. The models may be abstract state machines, or abstract data types, or algebraic axioms/postulates, etc. In other cases simplified models such as Petri nets are used. Sometimes the design is produced, then a separate recovery analysis is made, with appropriate design modification as necessary. Methods used in these recovery analyses include Failure Modes and Effect Analysis, Fault-Tree Analysis, and Mathematical Reliability Theory. Validation may now be done by verification (proving that the sequence of hierarchical structures begins with the functional specifications and ends with the design specifications, since each interface correctly simulates the action of the next higher), or by analysis of the simplified models and the affected design, or by simulation.

The area upon which most of the attention is traditionally lavished is that of transforming design specifications into the system design. The design is the implementation and documentation so that the system can be built using the support technology. The validation assertion is simply that the design satisfies the specifications. Dependable computer systems are built using a mixture of fault avoidance and fault tolerance techniques. Fault avoidance methods include: the use of reliable components (components chosen by physical fault analysis or reliability-oriented high-level language); components that are tested thoroughly; reliable interconnection techniques that are used and tested thoroughly; subsystems that are protected from expected interference; prediction of the system reliability; and systems tests that are devised to test the effectiveness of these procedures when a prototype or system is built. Fault tolerance techniques incorporate additional components and/or algorithms to ensure that occurrences of erroneous states will not result in later system failures. For example, if hardware is designed using the structural approach, then self-checking circuits will be used, together with circuits designed for ease of testing. If the functional approach is used, then a capability machine will probably be designed.

There are vexing design and validation problems. Are the algorithms valid in the presence of faults? How is reconfiguration authority granted? How can the algorithm and its implementation be tested?

Maintenance and change procedures must be devised so that their implementation does not invalidate an existing valid design, nor cause a revalidation of the total design.

VALIDATION TECHNIQUES

Validation techniques widely used are simulation and testing--but their lack of coverage renders them suspect for high-reliability applications.

Building and testing prototypes is necessary. The process is expensive, and devising a valid test procedure is very difficult. Verification, the use of mathematical proofs, is expensive but thorough. SRI is producing probably secure operating systems; IBM is validating combinational hardware design protocols. It is in this area of parallel, asynchronous operation that validation is most often found wanting.

After producing a prototype, validation should be able to assert that the prototype satisfies the design--is it the specifications? Difficulties arise in obtaining the right test equipment--and the effect of probes on measurements is well known. The coverage of fault insertion methods is small, and devising adequate tests is very difficult. The design is tested by structural tests while specifications are tested naturally by functional tests. Validating the effects of change without beginning again is a perennial difficulty.

After the prototype, production and maintenance have similar difficulties. They must assume that previous validation has discovered design and specification logical errors, and so must test for correct implementation of the design using tests. Structural tests are preferred, because of their diagnostic capabilities with faulty physical components. After modifications, the effects of these changes must be reflected in the design, its validation, in the tests, and in all documentation.

RESEARCH ISSUES IN SYSTEMS VALIDATION

The following four tasks should occupy our immediate attention:

1. Determine the relationships among known techniques used for validation: formal verification; simple specialized models; functional testing; structured testing. Determine a good combination of these which can be used. Begin with requirements.

2. Develop accurate models of physical faults in LSI. Also develop program reliability models. Use techniques such as mathematical reliability theory; failure modes and effect analysis; fault-tree analysis; fuzzy analysis and reliability theory; game playing; Walsh/Radamacher transforms, etc.

3. Try to develop a unified validation technique improving known methods.

4. Perform experimentation.

DESIGN FOR VALIDATION

John Wensley

INTRODUCTION

Digital systems by their very nature are much more difficult to validate than analog systems. This is primarily because they lack the following three properties which are extensively relied upon to validate analog systems:

1. Continuity
2. Inertia
3. Smoothness.

Traditional concepts of using testing, simulation, and emulation rely heavily upon these characteristics.

DESIGN

It is contended that design is a key element in the validation process. This includes both the process of design and also the end product of that process, namely the design itself. In validation we include analysis, testing, simulation, and formal proving.

SEGMENTATION

A key aspect of a design that assists validation is the creation of clearly defined boundaries or interfaces between the different parts of the system. Such interfaces include hardware, software, and operational aspects. If properly carried out, this segmentation of the system into a number of parts reduces the problem of validation to that of validating a number of small parts and the interaction among them. This results in the complexity of validation being significantly less than for highly amorphous and interconnected systems.

DESIGN PROCESS

There are two techniques in the design process that must be used if validation is to be achieved with economy and confidence. First, the design process must itself be segmented (though not necessarily in the same manner as the design itself). This segmentation of the design process is in terms of activities such as functional specifications, design specifications, implementation, and operational descriptions. Second, as each activity is carried out the products of that activity must be presented in a formal and rigorous manner. This includes specifications, drawings, programs, physical hardware, and the like.

SESSION II - PROGRAM (SOFTWARE) VALIDATION

Susan L. Gerhart, P. M. Melliar-Smith, and Herbert Hecht

SUMMARY

To examine the state of current knowledge of software validation, let us separate the aspects of software quality achievement, and assessment of that quality.

STATE OF CURRENT KNOWLEDGE

I. Software quality achievement

- Importance of hierarchy
  - Design
  - Specification
  - Validation chain
- Alternative validation methods
  - Testing
  - Proving
- Fault-tolerance framework
  - Recovery block

II. Software quality assessment

- Start on data collection, classification
- Complexity measures
- Primitive models.

Software engineering research has produced a host of techniques to improve software quality. The most important such concept is hierarchy of systems. Systems designed and implemented in (not necessarily identical) hierarchies provide natural validation boundaries at each hierarchical level. Each level of the hierarchy should be rigorously specified, with verification between levels used as part of the validation process.

Alternative methods of validation include testing and proof. Testing provides an experimental approach for improving and assessing software quality, but is limited by lack of underlying theory which permits stronger inferences from the experimental results. Proof provides analytical and mathematical evidence and explanation of how various properties are achieved. Neither testing nor proof is universally applicable and both are often unreliable and enormously expensive. Testing is more effective at the lower (module) levels, whereas proofs are more effective at higher levels.

A commonly accepted framework for achieving fault tolerance in software is the recovery block concept which permits alternate software to continue program operation when an error is detected in the main program path.

Assessment of quality is a less well developed area than achievement of quality. Overall, this area lacks data, good mathematical foundations, and experimental validation of concepts. However, a start has been made on the collection and classification of errors; there are also some measures for complexity of programs and some proposed primitive, but unsatisfactory, models for software reliability.

The needs for advancing the current state of knowledge in software validation may be classified into needs for new information and experiments as illustrated below:

NEEDS

I. Information and data

- Gross distribution of failures: hardware, software, human especially, severity
- Reliability data on how previous and current flight control software turned out, e.g., "only critical" observation
- Applicability and reliability of alternative validation methods

II. Case studies and experiments

- Simplified flight control system
  - Illustrate

    Hierarchy

    Specification
  - Capture essential ideas, but small enough to study, i.e., basis for research
  - Context for say, three more detailed excerpts

- Validation methods--practice and assessment
  - Redundant

    Applicability, reliability, cost
  - Integrated

    Applicability, reliability, cost
- Demonstration of recovery blocks
  - Acceptance tests
  - Correlated fault recovery
- "Model" model
  - Good theory
  - Practical.

With respect to information needs, there should be a catalog of at least a gross distribution of failures for general, highly reliable systems. More specifically, such data should be gathered on flight control systems motivated by various vague observations, such as "the only failures are critical ones." This catalog should include software, hardware, sensor, human failures, etc., along with a classification by severity. It is even conceivable that the applicability and reliability of the alternative validation methods will be illustrated by some existing data.

Some necessary experiments and case studies are evaluation of the reliability, applicability, and cost of various validation methods, with emphasis on their redundant and integrated use. Demonstration of the recovery block concept with emphasis on acceptance tests and correlated fault recovery could provide a significant advancement in the state of the art. A "model" model should be produced which achieves general agreement on its theoretical basis and practicality.

Software engineering research proceeds through the study of simplified versions of complex systems. Therefore, a simplified flight control system should be developed to illustrate hierarchy and specification issues and to capture and expose the essential problems and solutions involved in flight control systems. The system must be small enough for outsiders to study and should be directed toward research issues.

LIMITATIONS OF PROVING AND TESTING

Susan L. Gerhart

LIMITATIONS OF VALIDATION

As a framework for considering limitations of validation, let us consider three types of limitations applicable to any activity--what we know, what we can say, and what we can do:

GENERIC LIMITATIONS

- What we know
  - Theory } terms, techniques
  - Theories } general theorems
- What we can say
  - Gap between knowing and saying
  - Need for formal techniques
- What we can do
  - Time
  - Money
  - People
  - Motivation.

Ability to "know" and to "say" are not the same. For example, many people know one or more programming languages extremely well in that they can write, correct programs, spot language-related errors in other programs, and answer questions about the language. But few languages have had their semantics fully explicated or have even been well described in reference manuals. Knowing doesn't imply "ability to say."

With respect to computing system components and validation, we need to know (1) theory, (2) semantics of the system, (3) semantics of the problem addressed by the system, and (4) requirements and specifications to be met by the system. To some extent, it must be possible to say a great deal about each of these.

LIMITATIONS OF THE PROOF-ONLY APPROACH

Proving may be classified into broad categories as follows:

- Mathematical proofs
  - Lemmas, general theorems
  - Inductive methods
- Interactive verification tools
  - People state and structure theorems/proofs
  - System bookkeeps and simplifies.

The theory behind proving is well enough developed not to present a major limitation, although new problems may require some extensions. The mode of proving is determined by what we can say about semantics of system components and problems. The general techniques will work even if we cannot write down formal descriptions; everything is done informally with arbitration by people who know what is going on. But having precise language semantics, mathematical equations for problems, system component interfaces, etc., opens the way for either more formal nonmechanical proofs or for mechanical proofs. There is no reason why reasonably skilled, motivated people cannot carry out proofs to a great level of detail. Mathematicians do it, ancient astronomers did it, doctoral students do it--why not?

The state of the art in interactive verification tools is improving rapidly. Systems exist which can interactively produce proofs (with a reasonable division of labor between people and machines) and which can automatically prove some restricted classes of problems. Like all systems, these become large and cumbersome and must be tuned to user needs acquired through painful experience. This learning experience is going on now, but there is no reason to view even better human-engineered systems as a panacea. Theorems must be valid before they can be used in proof and an enormous amount of effort (and repeated failures) may go into finding the exactly correct form of the theorem. Furthermore, historical experience has taught us to beware of "proven" theorems: they may be based on faulty assumptions of logic and be incorrect.

The limits to proving can be outlined as follows:

- Theory
  - Invariants
  - Abstraction
- Theories
  - Data structure
  - Concurring problems
  - Paging behavior

- Semantics
  - Programming language
- Specification
  - Language and techniques
- Resources
  - Task requires enormous time
  - People require more expertise.

The proof-only approach primarily is limited by "what we can say." Years might go into developing appropriate and accurate formal definitions of languages. It may take months to state all the specifications and environment assumptions for a specific problem. The resulting statement may be too difficult for other people to understand.

Proof is also limited by "what we can do," mainly with respect to resources. A dedicated PDP-10 and a team of several people doing almost nothing but proofs is a realistic requirement. Skills probably exist or can be acquired through study equivalent to a few academic courses and some on-the-job experience. The most valuable contribution of such a system would be to keep the human participants "honest," challenged, and unburdened by repetitive tasks.

LIMITATIONS OF THE TEST-ONLY APPROACH

Testing may be classified into broad categories as follows:

- Experiment
  - Select data to reveal errors
  - Execute and evaluate
- Tools
  - Monitor coverage
  - Symbolic execution.

Testing shows a different limitation from proving, namely the absence of a well-developed theory. Such a theory must incorporate mathematics (logic and/or statistics), management (devotion of resources and planning), and psychology (use of intuition about sources of errors and recognition of human responses to errors). The deficiency of theory may be related to "what we can say," but conversations at testing conferences and sessions seem to indicate otherwise. Each

individual appears to have a weak, intuitive feeling for testing, but no universal consensus exists about what to test for or how to do it. Indeed, there is not even a standard set of definitions.

Paradoxically, a lot of testing can be done even without a theory or good semantics of the system and problem. The drive is to achieve some minimal level of test coverage (over the input or the structure of the system) and then to keep expanding the testing for further, different aspects. This type of activity alone can reveal an enormous number of problems. But the limitation is: what conclusions can you draw from having tested the system a certain amount? What has not been tested?

In comparison with proving, there are many people skilled in testing (at least for their specialties) and there is precedent for committing huge resources for testing. The problem is to direct those skills and resources toward more productive testing, but only a better theory can yield that direction.

The limitations of testing are outlined below:

- Theory — How to select and evaluate test data
- Theories
  - Errors
  - How to cover
- Semantics
  - Weak guides
- Specifications
  - Weak guides
- Resources
  - Coverage tools, test shops to achieve weak, broad goals
  - People have skills
  - Precedent for huge expenditures.

SUMMARY OF LIMITATIONS OF PROVING AND TESTING

Proving is limited by what can be said well enough to yield the basis for a proof. That pushes proving toward a level of abstraction where ideal or simplified environments and problem semantics must be considered so that proofs can be carried out reasonably completely. Testing, by contrast, is a very concrete activity which can be measured against some minimal standards of coverage,

but it cannot be extrapolated further than these minimal, concrete standards. For example, consider a simple program. Testing can show that all statements can be executed, but usually not that all expression values were computed and hence that all paths were "covered." Proving can show that for any arbitrary input the right output is obtained, assuming the definition of all programming language constructs and problem terminology. Testing can validate for some data, but not for all; proving can validate for all data, but may be wrong on some. These limitations are outlined as follows:

- Proving
  - Strong theory, strong demands
  - High levels of abstraction--simplified, idealized
  - Prone to errors (in proofs and assumptions)
- Testing
  - Weak theory, minimal demands
  - Concrete, detailed
  - Cannot infer, cannot select well.

Both yield insight and confidence; neither yields conclusive evidence.

COMBINATIONS OF PROVING AND TESTING

1. On critical components of significant complexity, proving and testing can both be used to their full extent, in the hope that one or the other will reveal any errors or deficiencies.

2. On uncomplicated or straightforward combinatorial problems where "all" can be tested by some standard of exhaustion, testing is clearly superior. But where algorithmic complication or nonalgorithmic combinatorial possibilities exist, proving is clearly superior.

3. A system may be produced at a sufficiently high level of abstraction such that proofs can be completely carried out. However, proofs for implementations of all these resources may require semantics and resources which are out of bounds. Proving at the abstract level, testing at the concrete level seems appropriate.

4. Some existing techniques and technology combine both testing and proving in the form of symbolic program execution. The testing limitation comes in from the inability to cover all paths of a program, whereas the proving limitation comes in with the need to simplify execution expressions and to determine feasibility of paths. But the comparable strengths also exist--more data is covered by dealing with symbolic data and hence more is learned from the testing.

5. The theory of both testing and proving is starting to show some examples of "proving-based testing" and "testing-based proving." The former involves theorems which state the conditions under which a characterized set of test data can demonstrate correctness. The latter uses testing to provide a large basis of results from which ad hoc inferences can be made.

6. The domain of a program may be partitioned into a small part exhaustively covered by testing and the rest covered by proving.

The following shows schematically the different combinations possible, as well as the strengths and weaknesses of some of these combinatorial components:

- Combinations
  - Split-up
    - Components
    - Domains
  - Parallel
    - Prove test
      - Prove
      - test
  - Serial
    - Testing + Proving
    - Proving + Testing
    - Testing and Proving
- Capabilities
  - Proving
    - Mathematical, analytic methods
    - Mechanical assistance
    - Spin-offs to language, design

- Testing

  Coverage tools

  Insight

  Early detection

- Needs
  - Testing theory - inference
  - Experience with proving
  - Simplified examples.

IMPLICATIONS OF FAULT-TOLERANT SOFTWARE

Certainly, fault tolerance can go a long way toward covering the limitations of the "can't say" variety or the multitude of possibilities which cannot all be covered by either testing or proving. Fault tolerance assumes that enough is known and can be said at certain program points to determine whether the current computations are plausible or will not lead to further errors. That is, fault tolerance is an add-on which may provide intermediate assumptions on which to base proofs or which may provide internal constraints to be satisfied, no more than that. However, the theory of proving does not include much address to fault tolerance, but this may be almost beyond any theory.

Testing also seems to have problems with respect to fault tolerance--how do you make things happen that are not supposed to happen? Perturbing the environment in various ways can accomplish some of this, but what is the full range of possible perturbations?

Additional materials submitted at the meeting are included in Appendix B.

FAULT-TOLERANT SOFTWARE

P.M. Melliar-Smith

The reliability requirements imposed on flight control software are very much greater than those current in other applications of computers. To meet the 10^{-9} probability of failure over a 1-hour flight requires a reliability about a million times greater than the very best that can be obtained from nontrivial programs by conventional means. Even the requirement for the 40-second autolanding sequence is about a thousand times beyond the current state of the art. It is impossible to verify by observation that these reliability levels have been attained, and thus certification must depend on some form of logical argument that will justify a reliability claim well beyond what can be measured. Such arguments may be viewed with suspicion, and rightly so, for while the argument might be logically sound in itself, it is necessarily based on assumptions which may be unjustified. The following summarizes some of the arguments used to prove SIFT reliability:

THE PROOF OF SIFT RELIABILITY

Problem statement,	Informal, MTBF, etc.,
Reliability model,	Markov model,
Requirement,	Predicate calculus,
Specification,	"Special" specifications,
Implementation,	Pascal program,
Operational system.	Bendix 930 code.

It is appropriate to examine the assumptions implicit in each of the available methods of obtaining reliability, program testing, program proof, and fault-tolerant design:

- Fault-tolerant software (recovery blocks)
  - The (system) design specification is correct
  - We can recognize errors
  - Some aspects of the specification of the computer
  - Alternative programs do not contain the same faults

- Testing in a simulated environment
  - The system specification is correct
  - The specification of the controlled equipment (and its simulation)
  - Sufficient test cases
  - We can recognize errors
- Flight test
  - The system specification is correct
  - Sufficient test cases
  - We can recognize errors
- Program proof
  - The system specification is correct
  - The specification of the controlled equipment
  - The specification of the computer/language
  - We can do the proof
  - The proof is sound.

It is apparent that each of these approaches depends on many assumptions, several of which are common to two of the approaches. However, the assumption as to the validity of the system specifications is the only assumption common to all methods.

We believe that an argument that the flight control software actually meets the reliability requirement will be more convincing and less exposed to unjustified assumptions if it is based on testing, proof, and fault-tolerant design. The validity of the system specifications will need to be substantiated.

The flight control functions themselves appear to be quite amenable to fault-tolerant implementation of the kind provided by recovery blocks, though no actual examples are available. Alternates should be easy to construct and the provision of good acceptance tests appears relatively straightforward though it may depend on improvements in the methods of specification of flight control functions. No implementational difficulties should arise and it is possible that even simpler techniques than recovery blocks might suffice for the flight control functions. Considerably greater difficulty is presented by the operating system that runs the flight control programs, providing scheduling, input-output, error detection and recovery,

and system reconfiguration. It will not be easy to devise either acceptance tests or recovery mechanisms for such functions through research proceeds.

SOFTWARE VALIDATION

Herbert Hecht

SOFTWARE VALIDATION METHODOLOGY

One way of validating software against a requirement of less than 10^{-9} or 10^{-10} critical failures/hour is to run that software in a representative environment over some multiple of 10^{-10} hours, and to demonstrate thereby that the probability of a critical failure is less than the desired limit. This is manifestly impossible for the SIFT or FTMP software.

If the demonstration can be carried out in an environment that is more stressful than that of typical use, the time required for validation can be shortened. This is equivalent to the accelerated test methods used for hardware. The work of Myron Lipow provides a theoretical basis for this approach but there are at present no calibrations available that permit assignment of specific stress multipliers for a software test. Also, the contraction of demonstration time achievable by this method is limited. In hardware it seldom exceeds one or two orders of magnitude. Nevertheless, some effort in calibrating stress factors for software appears warranted.

A further reduction of the validation period may be possible by obtaining agreement on the ratio of critical failures to total software failures. Obviously not every failure is critical in the sense of "preventing the continued safe flight and landing of the aircraft," and if complete absence of failures is demonstrated over some period of operation it may then be equated to critical-failure-free operation over a longer period, again perhaps by one or two orders of magnitude. A more specific definition of a critical failure will be necessary in order to benefit from this dichotomy.

If fault-tolerant software techniques are utilized, the validation can be partitioned into validation of the primary routines (and acceptance test) and validation of the alternate routines. If perfect coverage (of the acceptance test) and complete independence of failures in the primary and alternate routines are assumed, a square root reduction in the demonstration time required for validation would be possible. With reasonable assumptions about imperfect coverage and correlated failure modes the reduction in demonstration period is still very substantial. By combining all three approaches (stress factors, ratio of critical to total failures, and fault-tolerant software), validation by demonstration of 5,000 to 10,000 hours may be possible.

REQUIRED PROGRAM

a. Determine stress factors for various levels of test applicable to flight software. Possible sources: current aircraft inertial navigation

software, Titan III software, Delta Inertial Guidance software, Shuttle, ABM software

b. Determine ratio of critical to total software failures. Possible sources: all those listed under (a) plus Viking ground processing, FAA en route and flight service software.

c. Establish coverage factors for acceptance test and correlation of primary and alternate routine failure modes for fault-tolerant software (fault-tree analysis currently being investigated).

INTEGRATION OF SOFTWARE AND SYSTEM VALIDATION

In terms of function, software validation and system validation are largely synonymous and should be combined. Deliberate introduction of hardware faults can be used to stress both system and software and thus permit test acceleration. Partitioning of hardware can be used to establish an upper bound on module failure probability (e.g., with 10 memory modules active, each test hour is equivalent to 10 memory hours).

Additional materials submitted at the meeting are included in Appendix B.

SESSION III - DEVICE VALIDATION

Edward J. McCluskey and William Mann

SUMMARY

The principal conclusions of the Device Validation session are:

- Current industrial practice is acceptance of test components for approximately 3 percent AQL at a 95 percent confidence level for "stuck-at" faults.

  More complete testing based on general fault models is theoretically possible, but the cost of such effort would be uneconomical with current state-of-the-art techniques.

- The trend toward higher levels of integration (more components per chip) make the testing problem worse.

- Extensive work is underway to develop circuit design techniques for building devices more amenable to testing. Such "design for testability" should reduce testing cost and improve test thoroughness.

- System hardware validation requires design validation as well as quality verification. (Discussed in comments prepared by William Mann.)

- Production process control is a necessary adjunct to device validation.

During the discussion period, Ron Coffin pointed out that:

(1) Generally the customer is not willing to pay significantly for increased reliability; and

(2) testability has to be designed into the device.

Gerry Masson stated that UNIVAC is conducting a study for NASA on the detection rates of intermittent errors.

Bill Carter pointed out that all errors are intermittent and that the basic problem is fault pattern coverage. A study by Ball and Hardie showed that a large number of errors did not produce failures.

Sal Bavuso stated that intermittent faults may go unreported. Some airpline pilots are experiencing intermittent electronic equipment faults which are corrected or "go away" before being reported.

DESIGN FOR MAINTAINABILITY AND TESTABILITY

Edward J. McCluskey

SUMMARY

The benefits of improved maintainability are reduced equipment downtime and repair costs; this results in improved performance and lower life cycle costs. The benefits of reduced downtime include improved availability for realtime systems and increased system throughput. Reduced repair costs result from the use of fewer, less-skilled repair people, a smaller spare parts inventory, and less replacement of good components.

SYSTEM TESTING

Thorough system testing and diagnosis are vital for good maintainability. The desired test characteristics are high coverage (i.e., a large percentage of possible failures detected) and low cost.

The cost of testing is based upon the following cost elements:

- Input test pattern generation
- Storage of input test pattern and correct responses
- Length of time required to carry out the test.

Difficulties

The difficulties in achieving good system testing and diagnosis include the long computation time required for test pattern generation, the extensive memory space required for test pattern and response storage, and the long test times required by sequential circuits.

Approaches to Reducing Difficulties

Several of the following approaches may reduce the above difficulties:

- Increasing access to system via test point insertion
- Use of scan techniques to permit easier test generation and shorter tests with higher coverage
- Use of random or pseudo-random input test patterns
- Compression of test responses by transition counting or linear feedback shift register encoding

- Disabling feedback paths during testing to permit combination circuit techniques to be used for sequential circuits
- Special test reset or initialization capability
- Use of on-chip test logic
- Use of error correcting or detecting coder for input information with on-chip code checkers.

MICROELECTRONIC DEVICE VALIDATION

William Mann

INTRODUCTION

This paper discusses the microelectronic device validation from device design through device fabrication to production testing. The verification processes are discussed. The paper concludes with a brief discussion of the applicability of current device testing techniques to fault-tolerant designs.

DEVICE DESIGN

The device design cycle is shown in Figure III-1. The main features of this cycle are the logic equation generation and the use of a computer-aided design (CAD) system to generate the photo mask and the test pattern program.

DEVICE FABRICATION (WAFER PROCESSING)

Device production flow is shown in Figure III-2. The main elements of the device fabrication cycle are the process control feedback, parametric testing and process verification. In-process inspections include optical, physical, and electrical testing.

Parameters tested on completed wafers include MOS thresholds, threshold stability and oxide breakdown voltage. Process verification includes reliability monitoring and device performance monitoring.

PRODUCTION TESTING

Test procedures for production items depend upon what the customer wants and is willing to pay for.

Under the Rockwell production testing procedures, the most stringent tests with highest probability of rejection are conducted first, so device faults will be detected as early in test sequence as possible. Once a device has been rejected, no more time is wasted testing it. Device testing techniques used are listed in Figure III-3.

FAULT-TOLERANT DESIGN TESTING

The applicability of device testing techniques is summarized in Figure III-4. Three of the testing techniques are not applicable to testing fault-tolerant designs. The LSSD and component parts testing could be applied with good to fair results. Self-testing is also possible.

DEVICE DESIGN CYCLE

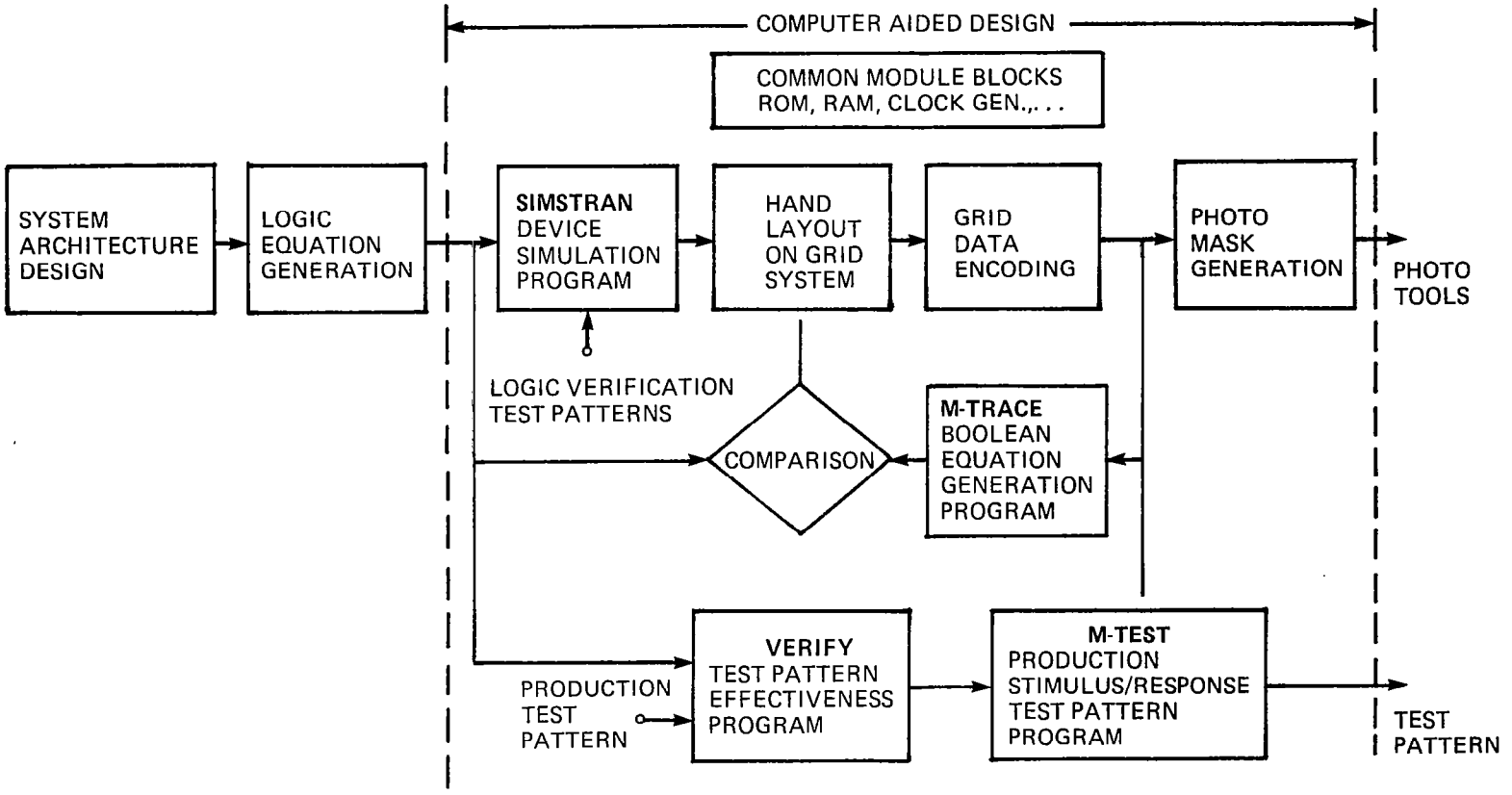


Figure III-1.- Device design cycle.

DEVICE PRODUCTION FLOW

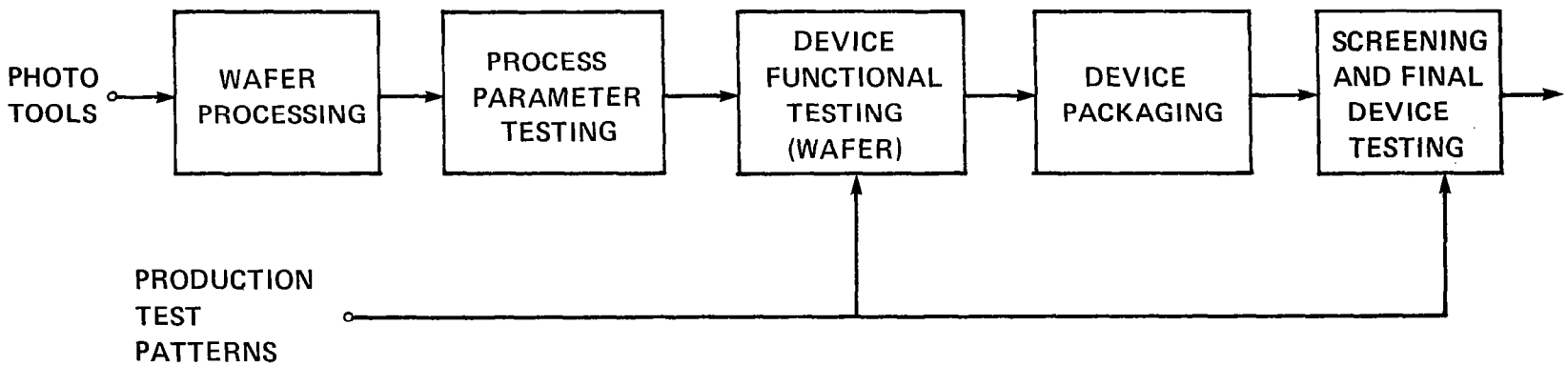


Figure III-2.- Device production flow.

DEVICE TESTING TECHNIQUES

- **FUNCTIONAL TESTING** – TEST THE DEVICE FOR ONLY THOSE FUNCTIONS IT IS SUPPOSED TO PERFORM
- **LOGIC TESTING** – TEST THE DEVICE AS A SET OF LOGIC EQUATIONS UNTIL SOME ACCEPTABLE TEST EFFECTIVENESS MEASURE IS ACHIEVED
- **LEVEL SENSITIVE SCAN DESIGN** – INSERTION OF PARALLEL TO SERIAL SHIFT REGISTERS WITHIN THE DEVICE TO DISABLE FEEDBACK LOOPS, SIMPLIFY, AND EXPEDITE LOGICAL TESTING AND FAULT DETECTION
- **COMPONENT PARTS TESTING** – TEST THE DEVICE BY INDEPENDENTLY TESTING THE MAJOR FUNCTIONAL COMPONENTS SUCH AS ROM, RAM, ALU, ETC.
- **SIGNATURE TESTING** – TEST CHARACTERISTICS OF PROPER DEVICE PERFORMANCE
- **SELF-TEST** – LET THE DEVICE EXERCISE ITSELF AND USE SOME FORM OF LSSD/SIGNATURE TESTING

Figure III-3.- Device testing techniques.

FAULT-TOLERANT DESIGN TESTING

TECHNIQUE	APPLICABLE
FUNCTIONAL	NO
LOGIC	NO
LSSD	GOOD
COMPONENT PARTS	FAIR
SIGNATURE	NO
SELF TEST	POSSIBLE

Figure III-4.- Fault-tolerant design testing.

SESSION IV - SAFETY, RELIABILITY, ECONOMICS AND PERFORMANCE ANALYSIS

J.J. Stiffler, Bob Flynn, and John F. Meyer

SUMMARY

The conclusions of this session can be summarized in the following five points:

1. Reliability modeling must be an essential part of validation in order to achieve a reliability level of a 10^{-9} probability of failure per hour.

2. A major limitation in the use of reliability models is the lack of data concerning device failure statistics and software failure statistics.

3. Success-path, fault-tree, and FMEA techniques can be used to determine the effect of low-level failures on system-level performance.

4. Emulation/simulation techniques are useful in determining reliability-model parameters.

5. Issues other than safety are also of concern, requiring modeling efforts that address questions of:

- Degraded performance
- Cost
- Maintenance
- Return on investment.

Summaries of each of the three presentations of this session follow.

LIMITATIONS OF RELIABILITY MODELING

J.J. Stiffler

Analytical reliability models have long been used as aids in the design of reliable systems. Traditionally, the congruence of the selected design with the model's predictions was validated by well-established statistical testing procedures. Such validation procedures are obviously not practical, however, when the system of concern is a fault-tolerant avionics system having an effective mean time before failure of over one million years! For such systems, the reliability model itself becomes an essential element in the validation process.

Some of the problems encountered in using analytic models to validate fault-tolerant systems include:

- Lack of important "raw" data (e.g., device failure statistics, device failure modes, failure correlations)
- Difficulties in developing an adequate coverage model
- Proliferation of the number of relevant system states needed in the reliability model (characterized by the number of failed elements of each type, the mission phase, the operation being executed at the time of the fault, the number of latent faults present, etc.)
- The significance of unanticipated failure modes
- The effect of cumulative round-off errors on numerical reliability predictions.

The talk concluded with a brief description of some of the modeling techniques being implemented in the CARE III Reliability Model currently under development for NASA's Langley Research Center.

The main points of the presentation are listed in Table IV-1, and a comparison of reliability modeling approaches is presented in Table IV-2. The points brought out concerning software reliability modeling reopened a discussion of the complexity of software vs. hardware reliability modeling. John Meyer commented that the distinction should not be between software and hardware faults but should be considered to be the distinction between "design" faults and "quality" faults. He felt that software modeling has a bad reputation because it has been based on hardware techniques and has been poor in quality. This fact, he continued, doesn't preclude the possibility of developing good software reliability modeling techniques in the future.

TABLE IV-1.- LIMITATIONS OF RELIABILITY MODELING.

LACK OF GOOD STATISTICAL DATA

- DEVICE HAZARD RATES
- TRANSIENT STATISTICS
- INTERMITTENT FAULT STATISTICS
- COVERAGE PARAMETERS.

LARGE NUMBER OF STATES (OPERATIONAL MODES) NEEDED TO DESCRIBE SYSTEM BEING MODELED. HAZARD RATES AND SYSTEM FAILURE RESPONSES MAY DEPEND UPON:

- NUMBER OF PRIOR FAILURES OF EACH RELEVANT CATEGORY
- STATUS OF PREVIOUSLY FAILED UNITS (FAILURE ACTIVE BUT UNDETECTED, FAILURE PASSIVE, ETC.)
- OPERATION BEING PERFORMED.

NUMERICAL ANALYSIS PROBLEMS

- ROUND-OFF ERRORS
- ILL-CONDITIONED TRANSITION MATRICES.

THE PROBLEM OF IDENTIFYING ALL "IMPORTANT" FAILURE MECHANISMS

SOFTWARE RELIABILITY MODELING CONSIDERATIONS

- "MODEL" IMPLIES DESCRIPTION OF FAILURES AS STOCHASTIC PROCESSES
- HARDWARE FAILURE MODELING HAS BEEN SOMEWHAT SUCCESSFUL WITH REGARD TO PHYSICAL FAILURES IN PREVIOUSLY FUNCTIONING DEVICES. LESS EFFORT HAS BEEN EXPENDED IN MODELING DESIGN OR SPECIFICATION ERRORS
- ALL (?) SOFTWARE FAILURES FALL INTO THE DESIGN OR SPECIFICATION ERROR CATEGORY
- FROM THE RELIABILITY MODEL POINT OF VIEW, THE EXISTENCE OF A SPECIFICATION OR DESIGN ERROR IS A RANDOM VARIABLE: ITS MANIFESTATION IS A RANDOM PROCESS.

TABLE IV-2.- COMPARISON OF ALTERNATIVE RELIABILITY MODELING APPROACHES

METHOD	ADVANTAGES	DISADVANTAGES	FIDELITY/COST RATIO
ANALYTICAL MODELS	RESULTING EXPRESSIONS MAY PROVIDE CONSIDERABLE INSIGHT INTO SYSTEM	CONCEPTUALLY COMPLEX FOR NONTRIVIAL CONFIGURATIONS. EXPRESSIONS MAY BE TOO CUMBERSOME WHEN SYSTEM HAS MANY OPERATING MODES	VERY HIGH FOR SYSTEMS WITH FEW OPERATING MODES
MARKOV MODELS (TIME DISCRETE)	CONCEPTUALLY STRAIGHTFORWARD	LARGE NUMBER OF STATES NEEDED TO DESCRIBE COMPLEX SYSTEMS. APPROXIMATION ERRORS MAY LIMIT FIDELITY	HIGH IF NUMBER OF STATES NOT TOO LARGE AND IF TRANSITION RATES NOT WIDELY DIFFERENT
MARKOV MODELS (TIME CONTINUOUS)	CONCEPTUALLY STRAIGHTFORWARD--RESULTING ANALYTICAL EXPRESSION MAY PROVIDE CONSIDERABLE INSIGHT	NUMERICAL ANALYSIS PROBLEMS FOR OTHER THAN "PURE DEATH" MARKOV PROCESSES	VERY HIGH FOR "PURE DEATH" PROCESSES
HYBRID MODELS	POTENTIALLY COMBINES BEST FEATURES OF ANALYTIC AND MARKOV MODELS. MAY BE MORE BROADLY APPLICABLE THAN EITHER	NONTRIVIAL RELATIONSHIP BETWEEN SYSTEM DESCRIPTION AND MODEL	REASONABLY HIGH FOR WIDE RANGE OF SYSTEM CONFIGURATIONS AND FAULT MODELS
EMULATION/SIMULATION	GOOD DESIGN TOOL. USEFUL IN EVALUATING FAULT RESPONSES	COST (ONE STATISTICAL SAMPLE PER RUN)	LOW (FOR RELIABILITY ESTIMATION)

RELIABILITY MODELS

Bob Flynn

Techniques for reliability analyses can be classified into three major categories, along with an additional auxiliary or supporting technique. The techniques are: success-path analyses, fault-tree analyses, and state-tree analyses. Simulation comprises the auxiliary technique.

These techniques may be described briefly as follows. In a success-path analysis, we describe the connections or relations that must exist between various elements of the system to achieve "success," i.e., the desired favorable outcome, as illustrated in Figure IV-1.

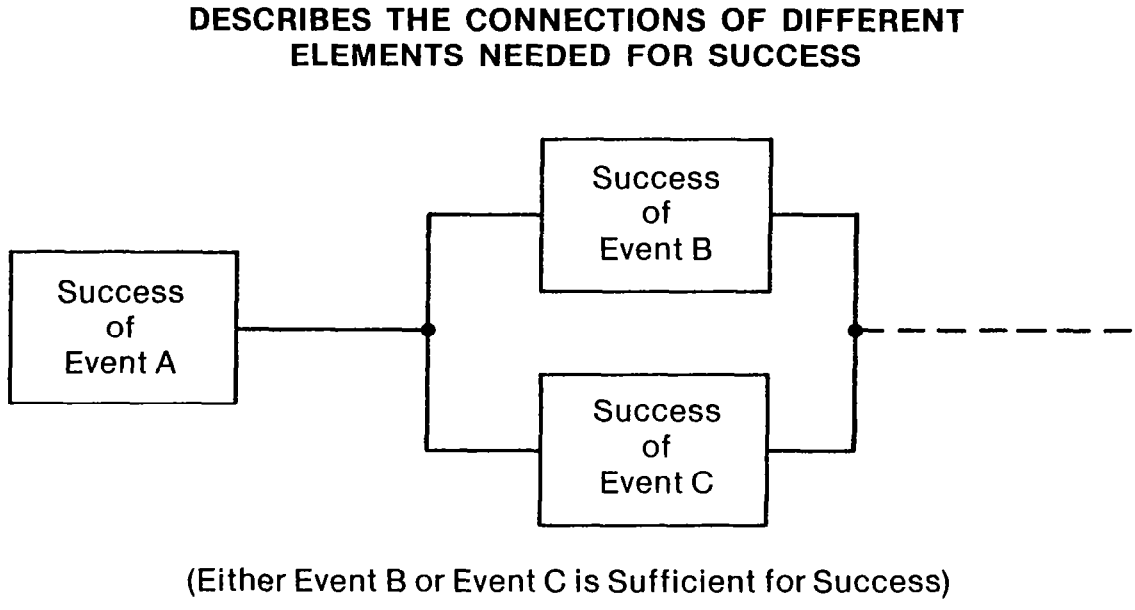


Figure IV-1.- Success-path analysis.

Fault-tree analysis starts from the opposite perspective. Specifically, a fault tree displays the ways in which failures of the various elements can combine and result in some "top level" disastrous fault, as in the example of Figure IV-2.

In the state-tree approach we divide the system into "stages," where a stage is a set of identical, redundant "modules." A module is the smallest functional entity that is considered in the analysis. The operational status of each stage is modeled by a finite order Markov process. Operational status is characterized by terms such as "no failures," "one failure," etc., down to "complete stage failure."

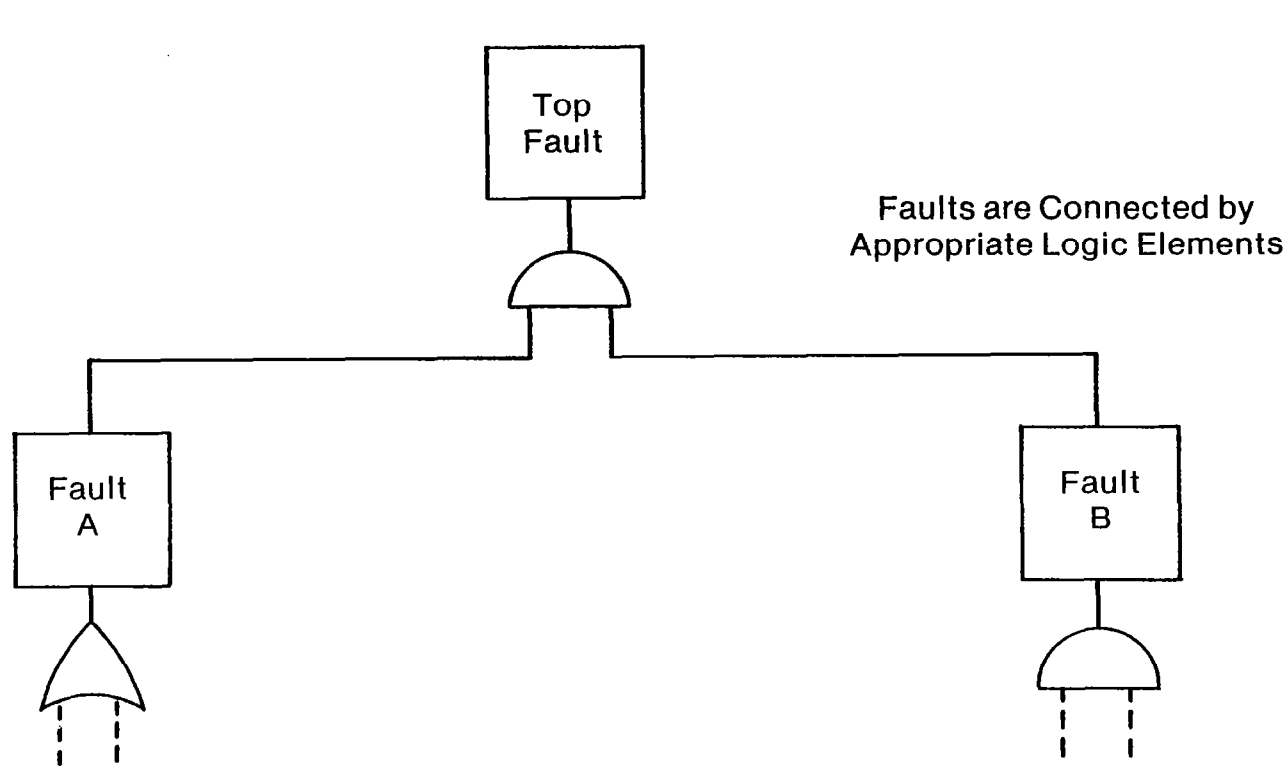


Figure IV-2.- Fault-tree analysis.

The Markov process is merely a set of first order, matrix differential equations. The coefficients of the matrix are transition rates (or failure rates). Figure IV-3 illustrates a Markov model.

The various reliability models have different characteristics. For sufficiently simple problems identical answers can be obtained from the various techniques. This is not necessarily true for complicated systems for several reasons. First, computer round-off error or other numerical problems can become significant for large models. Second, programs capable of handling large systems typically employ various mathematical approximations to calculate equations. Third, programs implementing various reliability model calculations are based on similar, but not identical, assumptions. It would be fortuitous if identical approximations and assumptions were made in different programs.

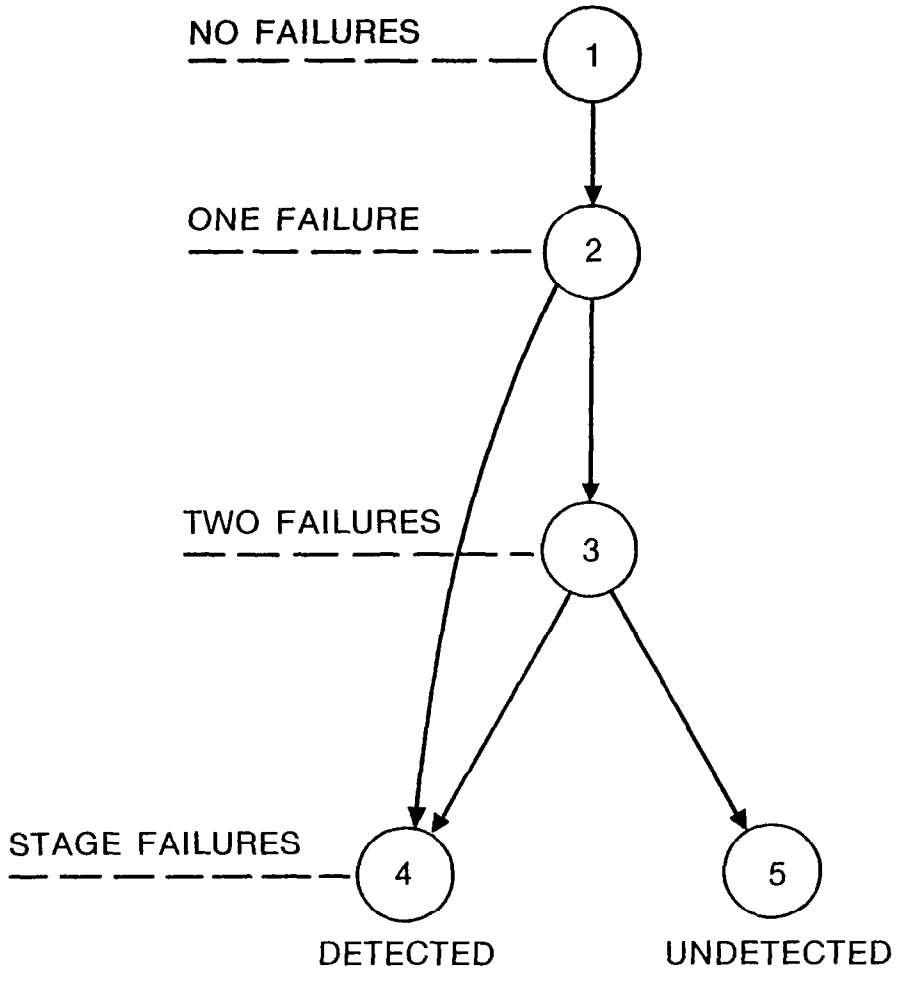


Figure IV-3.- Example of Markov model of a triplex stage.

Some further characterizations of the various approaches are:

1. Fault trees are good for describing a system to identify critical elements. (Herb Hecht noted that they also tended to surface correlation problems.)

2. The success path technique requires that all success paths be identified; consequently it tends to become more burdensome in fault-tolerant systems with many success paths.

3. The state-tree approach is probably the most versatile technique but detailed knowledge of the system is needed to formulate the correct Markov model.

The following conclusions follow from the above:

- The approach used in reliability analysis is often dictated by the personal preference of the analyst rather than strictly technical factors. This is, of course, a common phenomenon in other areas of analysis.

- No existing tool (computer reliability program) is adequate for all problems of interest.

- All of the approaches can result in a significant batch processing burden for complicated systems. This fact must be accepted as simply one of the costs of designing fault-tolerant systems. However, a modified state-tree Markov model approach with partitioning that results in very large, but sparse matrices might mitigate the computation burden.

PERFORMABILITY

John F. Meyer

The need for unified performance/reliability (performability) models, which use performance variables rather than a yes/no, failed/operational discrete state as the basis for calculations, was reviewed. The essential ingredients of such models were summarized, including the notions of base mode performance model, capability function, and model hierarchy. The basic points on these topics are presented in Table IV-3.

There was also a brief discussion of how performability models relate to more traditional analytic models that are typically used for computer performance and reliability evaluation. This relationship is illustrated in Table IV-4.

The problem of model solution (i.e., model-based performability evaluation) was then addressed and existing solution techniques were reviewed. Application experience with both modeling and solution methods were also reviewed. These model solution techniques are summarized in Table IV-5.

Finally, as a lead into the discussion period, needs for further development were outlined along with the roles that performability evaluation might play in the specification, design, and evaluation of fault-tolerant systems. These development needs include:

- More experimentation with current techniques
- More representative base models
- Improved solution techniques
- Approximation methods
- Interface with economic models
- Assessment of extent to which simulation and measurement can be used in performability evaluation.

The discussion of improved/combined modeling techniques brought out the problem of temporal dependency among states, a relationship not handled well by fault-tree techniques. John Myers observed that, more and more, the need for time-dependent analysis was surfacing.

John Meyer answered that in the specification phase, more must be thought of than 10^{-9}, "all or nothing" performance; that degraded operations must also be considered.

In a more general comment, John Wensley asked if it was intended to embed the performability model in an economic model, and was answered in the affirmative.

TABLE IV-3.- THE NEED FOR AN INGREDIENTS OF A UNIFIED PERFORMANCE/RELIABILITY (PERFORMABILITY) MODEL.

NEED

- Fault-tolerant avionic systems are multifunction systems which, typically, do not exhibit "all or nothing" performance.
- Instead, system performance is "degradable," i.e., depending on the history of the system's structure, internal state, and environment during use, one of several worthwhile levels of performance may be realized.
- If performance is degradable, traditional views of
  - Performance
  - Reliability

  need to be unified to permit quantification of the system's "ability to perform."

INGREDIENTS

- Base model--a stochastic model representing the dynamics of the system's structure, internal state, and environment.
- Performance variable--a random variable whose value is uniquely determined by the state trajectory (sample path) of the base model process.
- Capability function--the function that associates with each state trajectory its corresponding level of performance.
- Model hierarchy--hierarchical elaboration of the base model to simplify the formulation of capability.

TABLE IV-4.- RELATION OF PERFORMABILITY MODELS

TO PERFORMANCE, RELIABILITY MODELS.

	PERFORMANCE	RELIABILITY
BASE MODEL	FIXED STRUCTURE	FIXED INTERNAL STATE, ENVIRONMENT
PERFORMANCE VARIABLE	PERFORMANCE MEASURE OR INDEX	TWO-VALUED VARIABLE--SUCCESS, FAILURE
CAPABILITY FUNCTION	NOT USED EXPLICITLY	STRUCTURE FUNCTION
MODEL HIERARCHY	NOT USED EXPLICITLY	FAULT-TREE

TABLE IV-5.- MODEL SOLUTION TECHNIQUES.

MODEL SOLUTION

Let

U = trajectory space of base model

A = levels of accomplishment (value set of the performance variable)

γ: U → A, the capability function.

Model solution is basically a two-step procedure.

1) For each a in A, determine

$$U_a = \gamma^{-1}(a)$$

2) For each U_a, determine

Prob (U_a)

SOLUTION TECHNIQUES

Step 1)

- Decomposition of γ into interlevel translations X_i
- Top-down computation of $\gamma^{-1}(a)$, beginning with $X_0(a)$

Step 2)

- Phased base models
- Iterative computation (beginning with phase 1) of Prob (U_a)

SESSION V - FAULT MODELS

John Myers

SUMMARY

INTRODUCTION

Only one speaker delivered prepared comments during Session V. These prepared comments were intended to discuss five apparently disjointed aspects of fault models and not to present a comprehensive review or critique of the state of the art in fault models. Additionally, there was no time for a formal discussion of the prepared comments. Thus, this summary presents the result of numerous informal comments and particularly the conversations among Gerry Masson, John Myers, Bill Rogers and Steve Toth.

PRACTICAL THOUGHTS ON FAULT MODELS

Discussion of the general state of the art in fault modeling and the particular characteristics of validating avionics for commercial passenger transport aircraft prompted many comments. These are (somewhat arbitrarily) presented under the following three headings:

- Generalities not to be forgotten
- Comparing the safety modeling of fault-tolerant avionics with the modeling of special weapons handling
- Specific suggestions.

GENERALITIES NOT TO BE FORGOTTEN

1. Modeling is the formulating of things that are important to you--therefore, if your purpose changes, your model must change. General-purpose modeling (including general-purpose simulation) is useless.

Example: Very different simulations are required to support:

- The study of the long-term effects of preventive maintenance
- The study of the short-term effects of random failures
- Diagnosis of intermittent faults.

2. If you don't understand it, you can't model it. Probabilities make sense only after a structure of what can happen, at various times and places, is established. Such a structure can be established only with regard to a closely controlled environment. Therefore, a high degree of safety can be guaranteed only for such an environment; therefore, fault-tolerant computers, like atom bombs, can be handled safely only if the environment is closely controlled.

3. Failure can be defined only as a departure from functioning; thus functioning must be defined first.

4. Some questions can be modeled by novices but not by experts: the expert (but not the novice) sees that the question fails to define adequately a process into which the modeler can enter to obtain the essential insight concerning "what matters." Master modelers are masters of how to break down a system conceptually into parts that have:

- Understandable engineering and operational characteristics
- Tractable mathematical behavior.

5. Once the modeling purpose is well defined for some issue of fault-tolerant computers, it would be very beneficial to draw on the talent of people who, with an investment of twenty years and millions of dollars, have developed working models that now guide (and are corrected by) practical operations in real organizations.

COMPARISON WITH SAFETY MODELING OF WEAPONS HANDLING

The modeling of safety of aircraft using fault-tolerant computers for flight-critical functions was compared with the modeling of safety of the handling of atomic weapons. The questions asked in the two cases turn out to be strikingly different.

Case of fault-tolerant computers: "Will they be safe enough to justify the commitment of dollars to the next step of development?"

Case of weapons handling: "Given that the country is deploying them, how can we do it most safely?"

The second form of question promotes more fruitful analysis because it entails a practical and concrete context. Addressing it has led to many demonstrable improvements in safety. An actual, operating machine is required for the weapons-style analysis.

SPECIFIC SUGGESTIONS

Concern for validation should not stop with certification, but also should include long-term use. Modeling should work backwards from a hypothetical aircraft crash, so as to guide:

- Formulation of the investigation of a crash
- Requirements for a survivable recorder of the computer's last minutes
- Design requirements of the computer so that it can be guaranteed to leave an "audit trail" in the recorder.

FAULT MODELS

John Myers

INTRODUCTION

I shall discuss the following five aspects of faults and fault models:

1. X-rays were discovered because of the "faulty" storage of photographic film.

2. The continued evolution of bugs and bug catchers is forced by their environment.

3. A card file model can help to establish the conceptual boundary between the foreseen and unforeseen.

4. Structural uncertainty is a fundamental property of systems.

5. Some fundamental issues concerning known faults must be addressed:

- The What of 0 and 1
- The When of 0 and 1
- The Where of 0 and 1
- Fanout
- Glitch
- Concurrency.

Furthermore, an attempt shall be made to show the relationships among these five apparently disjointed items.

DISCOVERY

Discovery is by its very nature unpredictable. For example, X-rays were discovered because photographic film was improperly stored in a cabinet along with other research materials. Coincidentally, someone was experimenting with radium and happened to store the radium in the same cabinet. The film got cloudy; the source of the clouding was traced to the radium, which appeared to emit previously unknown radiation. Henceforth, it was expected that radium produces X-rays.

BUGS AND BUG CATCHERS

Frogs catch bugs; they evolve ever more efficient mechanisms and techniques for catching bugs. Soon all of the bugs would be eaten and all the frogs would starve to death. However, fortunately for the bugs (and of course frogs), the bugs evolve more efficient mechanisms and techniques for avoiding frogs. Thus, the balance of the system is maintained.

Similarly, it's the nature of our business to look beyond our current boundaries and to demand more than we can do now. Examination of the hydrogen spectra led to the construction of the Bohr atom. Playing with the Bohr model and careful examination of the spectra led to discovery of the fine structure. The experimenters, however, still were not satisfied and further investigation led to discovery of hyperfine structure.

We should recognize that as soon as we learn how to make something "good" someone will use our good thing in a way that requires it to be better. Specifically, as soon as "all" faults are removed from an item, it will be used in such a way that the undiscovered faults become important. Perhaps we are approaching such a state when we consider the ultra-reliable systems of fault-tolerant avionics.

Note that the unforeseen fault is different in kind from the known fault of low (predicted) probability. Contrast the following two situations:

1. In a foreseen event of low probability a cable that is nominally adequate to support a load of 1000 pounds breaks while holding only 100 pounds. If this happens once in a long run of tests, the engineer merely notes that a low-probability event occurred; he may not even have to change his estimate of the probability. He expects a repetition of the experiment to result in the likely, not the unlikely, outcome: next time the cable will hold the load.

2. As an unforeseen event consider the situation of Christopher Columbus. While searching for a route to the Indies, he bumps into America. He changes his thinking; he expects that in a repetition of his voyage he would again encounter the same previously unforeseen America.

CARD FILE MODEL

I offer you a perspective in which all faults are classified into one of two categories: the foreseen, and the unforeseen. I would like to offer you a highly conceptualized model to be used in this perspective. It is a (hypothetical) card file model to help us think about the relationship between the foreseen but unexpected and the unforeseen or completely "surprise" aspects of fault occurrence during validation. This card file model, which forms the boundary between the foreseen and unforeseen, is a cross-indexed filing system which consists of descriptors, items of content, pointers, and historical activity log.

When a fault event occurs, it will be foreseen or unforeseen, If foreseen, then its descriptors will already be on the descriptor list. Too many such faults may threaten the validity of the system, but the occurrence of such a fault per se does not. But not all faults are foreseen: a fault can occur for which no adequate descriptors exist. In this case the engineer creates new descriptors and enters these on the descriptor list, and then records the fault in the file.

The existence of unforeseen faults threatens the validity of the system. When such ad hoc entries accumulate in the file, one must question the hypothesis--i.e., the fault model--that underlies the design of the system. This question implies a crisis. Either the engineers will, through a creative act, reorganize the file and show that it fits an improved hypothesis, or else they will fail to show that the system is valid. If the creative reorganization succeeds, then a new fault model is produced.

STRUCTURAL UNCERTAINTY

However, our rudimentary fault model is limited by its inherent nature, not the least limitation of which is the Structural Uncertainty Principle. Consider the following:

An operating computer system is distributed in space and time. To measure its state at any instant, we must first collect in one location all the register states from different points in the computer. So we connect wires to the registers and bring them to our display board. But it takes an indeterminate time for the signals to propagate through the wire. Therefore, state transitions which appear to coincide at the display board actually occurred at different times within the computer. In fact, state transitions we consider acceptable at the display board may in fact be "glitches" and "faults."

Hence the Structural Uncertainty Principle states:

1. There is a definite, absolute limit on our ability to make measurements on a system. We cannot measure with certainty (either individually or together) the state and transitions from state to state of a system.

2. Fault models assume the existence of a known state structure which cannot be determined.

3. The fact that a system is put into a state where an unforeseen transition (to an unforeseen state) might occur does not guarantee that transition will indeed occur. Nature can withhold surprises or give them out later.

4. Unforeseen faults correspond to unforeseen structure, or lack of structure, in a system or component. The unforeseen fault is different in kind from a known fault of low probability.

STATES AND FAULTS

The concept of "state" was explored at the micro level where special interesting types of "0" and "1" faults were discussed, as noted in Table V-1, and at the macro level, as noted in Table V-2.

TABLE V-1.- THE "WHAT," "WHERE" AND "WHEN" OF "0" AND "1."

1. What are the states? Shottky TTL exhibited stable "½" (neither "0" nor "1") under very special conditions.
2. "0" and "1" Where? Special separation matters; fanout and cross-talk blur physical boundaries.
3. "0" and "1" When? Glitches and lack or presence of signal concurrency may be causing unexpected failure in a state.

TABLE V-2.- CONCURRENCY OF GLITCH.

The concept of state, for a large system, becomes less usable as increased reliability is required because:

1. One needs to be able to measure states at the rate a system generates them. Slowing down the clock may eliminate possible races, glitches, etc.
2. The measurement of a state requires assembling signals from separated origins at a common location. We change the nature of the beast merely by inserting our measuring probe.
3. The signals are dynamic and stopping them may destroy the evidence.
4. The question of which signal values go with which state (to or from which state) must be decided by an arbiter. Arbiters have to be located in the system or in our probe.
5. Arbiters are "glitchy": increased reliability of an arbiter demands a reduction in rate of measurement to allow time for the arbiter to reach and communicate a decision.

APPENDIX B - ADDITIONAL MATERIALS SUBMITTED BY SUSAN L. GERHART

AND HERBERT HECHT

LIMITATIONS OF PROVING AND TESTING

Susan L. Gerhart

As a framework for considering limitations of validation, let us consider three types of limitations applicable to any activity - what we know, what we can say, and what we can do. Ability to "know" and to "say" are not the same. For example, many people know one or more programming languages extremely well in that they can write correct programs, spot language-related errors in other programs, and answer questions about the language. But few languages have had their semantics fully explicated or even been well described in reference manuals. Knowing doesn't imply "ability to say."

With respect to computing system components and validation, we need to know (1) theory, (2) semantics of the system, (3) semantics of the problem addressed by the system, and (4) requirements and specifications to be met by the system. To some extent, it must be possible to say a great deal about each of these. To do validation, we must have skills, tools, and resources (time, personnel, money).

LIMITATIONS OF THE PROOF-ONLY APPROACH

The theory behind proving is well enough developed not to present a major limitation, although new problems may require some extensions. The mode of proving is determined by what we can say about semantics of system components and problems. The general techniques will work even if we cannot write down formal descriptions; everything is done informally with arbitration by people who know what is going on. But having precise language semantics, mathematical equations for problems, system component interfaces, etc. opens the way for either more formal non-mechanical proofs or to mechanical proofs. There is no reason why reasonably skilled, motivated people cannot carry out proofs to a great level of detail. Mathematicians do it, ancient astronomers did it, doctoral students do it - why not?

The state of the art in verification tools is improving rapidly. Systems exist which can interactively produce proofs (with a reasonable division of labor between people and machines) and which can automatically prove some restricted classes of problems. Like all systems, these become large and cumbersome and must be tuned to user needs, acquired through painful experience with them. This is going on now, but there is no reason to view even better human engineered systems as a panacea. The limitation is that the theorems must be valid before they can be proved and an enormous amount of effort (and repeated failures) may go into finding the exactly correct form of the theorem. Such a system is a valuable resource, if for no other

reason than keeping the human part of the system honest, unburdened by repeated writing, challenged, and interested.

The main limitation to the proof-only approach is in the "what we can say" category. Years might go into developing appropriate and accurate formal definitions of languages. It may take months to state all the specifications and environment assumptions for a specific problem. The resulting statement may be too difficult to understand by other people. A second limitation is in "what we can do," mainly with respect to resources. A dedicated PDP-10 and a team of several people doing almost nothing but proofs is a real possible requirement. Skills probably exist or can be acquired through study equivalent to a few academic courses and some on-the-job experience.

LIMITATIONS OF THE TEST-ONLY APPROACH

Testing shows a different limitation from proving, namely the absence of a well-developed theory. Such a theory must incorporate mathematics (logic and/or statistics), management (devotion of resources and planning), and psychology (use of intuition about sources or errors and recognition of human responses to errors). The deficiency of the theory may be related to "what we can say," but conversations at testing conferences and sessions seem to indicate that each individual has a weak, intuitive feeling for testing, but that no universal consensus exists about what to test for or how to do it. Indeed, there isn't even a standard set of definitions.

Paradoxically, testing is the type of validation where a lot can be done even without a theory or good semantics of the system and problem. The drive is to test for some minimal level of coverage (over the input or the structure of the system) and then to keep testing for further, different aspects. Just this type of activity can reveal an enormous number of problems. But the limitation is: what conclusions can you draw from having tested the system a certain amount? What hasn't been tested?

In comparison with proving, there are many people skilled in testing (at least for their specialities) and there is precedent for committing huge resources for testing. The problem is to direct those skills and resources toward more productive testing, but only a better theory can yield that direction.

SUMMARY OF LIMITATIONS

Proving is limited by what can be said well enough to yield the basis for a proof. That pushes proving toward a level of abstraction where ideal or simplified environments and problem semantics are considered so that proofs can be carried out reasonably completely. Testing, by contrast, is a very concrete activity which can be measured against some minimal standards of coverage, but it cannot be extrapolated further than these minimal, concrete standards. For example, consider a simple program. Testing can show that all statements can be executed, but usually not that all expression values were

computed and hence that all paths were "covered." Proving can show that for any arbitrary input the right output is obtained, assuming the definition of all programming language constructs and problem terminology. Testing can validate for some data, but not for all; proving can validate for all data, but may be wrong on some.

COMBINATIONS OF PROVING AND TESTING

1. On critical components of significant complexity, proving and testing can both be used for their full extent, in the hope that one or the other will reveal any errors or deficiencies.

2. On uncomplicated or straightforwardly combinatorial problems where "all" can be tested by some standard of exhaustion, testing is clearly superior. But where algorithmic complication or non-algorithmic combinatorial possibilities exist, proving is clearly superior.

3. A system may be produced at a sufficiently high level of abstraction that proofs can be completely carried out. However, proofs for implementations of all these resources may require semantics and resources which are out of bounds. Proving at the abstract level, testing at the concrete level seems appropriate.

4. Some existing techniques and technology combine both testing and proving in the form of symbolic program execution. The testing limitation comes in from the inability to cover all paths of a program, whereas the proving limitation comes in with the need to simplify execution expressions and to determine feasibility of paths. But the comparable strengths also exist - more data is covered by dealing with symbolic data and hence more is learned from the testing.

5. The theory of both testing and proving is starting to show some examples of "proving-based testing" and "testing-based proving." The former involves theorems which state the conditions under which a characterized set of test data can demonstrate correctness. The latter uses testing to provide a large basis of results from which ad hoc inferences can be made.

6. The domain of a program may be partitioned into a small part exhaustively covered by testing and the rest covered by proving.

IMPLICATIONS OF FAULT-TOLERANT SOFTWARE

Certainly, fault-tolerance can go a long way toward covering the limitations of the "can't say" variety of the multitude of possibilities which can't all be covered by either testing or proving. This assumes that enough is known and can be said at certain program points to determine whether the current computations are plausible or will not lead to further errors.

The theory of proving does not include much addressing fault-tolerance, but this may be almost beyond any theory. That is, fault-tolerance is an add-on which may provide intermediate assumptions on which to base proofs or which may provide internal constraints to be satisfied, no more than that.

Testing seems to also have problems with respect to fault-tolerance - how do you make things happen that are not supposed to happen? Perturbing the environment in various ways can accomplish some of this, but what is the full range of possible perturbations?

FAULT-TOLERANT SOFTWARE

Herbert Hecht

ABSTRACT

Limitations in the current capabilities for verifying programs by formal proof or by exhaustive testing have led to the investigation of fault-tolerance techniques for applications where the consequence of failure is particularly severe. Two current approaches, N-version programming and the recovery block, are described. A critical feature in the latter is the acceptance test, and a number of useful techniques for constructing these are presented. A system reliability model for the recovery block is introduced, and conclusions derived from this model that affect the design of fault-tolerant software are discussed.

INTRODUCTION

The fault-tolerance features for software described here are primarily aimed at overcoming the effects of errors in software design and coding. In fortuitous circumstances they may also circumvent failures due to hardware design deficiencies of malfunctions in input channels. Fault-tolerance for random computer failures is outside the scope of the present discussion. In practice, the user wants computer system fault-tolerance, i.e., toleration of failures due to all causes, and this can be provided by a combination of established hardware fault-tolerance techniques (1) and the fault-tolerant software described here.

It is generally recognized that even very carefully designed and manufactured computer components may fail, and hardware redundancy is therefore provided in applications where interruptions of service cannot be tolerated. That software may fail is also widely recognized, yet this seems to have been perceived as a temporary shortcoming: today's software contains design and coding errors but it is expected that these will be eliminated once improved development and test methodologies or efforts at formal verification become fully effective. Deliberate use of redundant software to tolerate software faults is not an established practice today.

It is expected that the many efforts currently underway to improve software quality and reliability will indeed reduce failures but will not completely eliminate them. A number of thoughtful articles have pointed out the limitations of current test methodology (2) and of formal verification (3,4,5). Sometimes it is believed that maturity can provide freedom from software errors but that is not borne out by the experience on extensively used operating systems (6). For the foreseeable future it must then be expected that even the most carefully developed software will contain some faults, probably of a subtle nature such that they were not apparent during

software and systems testing. These faults will manifest themselves during some unusual data or machine state and will lead to a system failure unless fault-tolerance provisions are incorporated in the software. In the Bell Laboratories' Electronic Switching Systems (which employ hardware redundancy and thoroughly tested software) such software faults accounted for approximately 20 percent of all failures (7).

Fault-tolerance always involves some redundancy and therefore increases the resource expenditure for a given function. Software fault-tolerance of the type described here is therefore primarily aimed at applications where the consequences of failure are particularly severe, e.g., advanced aircraft flight control systems (8,9), air traffic control programs, and nuclear reactor safety systems. Sometimes fault-tolerance applied to a small segment of a large software system can provide a hardened kernel that can then be used to organize the recovery from failures in other segments (10). Such applications are today under study or in early development. Experience with these techniques in an operational environment has not yet been reported.

A survey of current approaches to software fault tolerance is presented in the next section. This is followed by a discussion of a particularly critical component, the acceptance test that determines when the primary software routine has failed. In the final section a system reliability model for fault-tolerant software is described.

CURRENT APPROACHES

Two different techniques for achieving fault tolerance in software have been discussed in the recent literature: the recovery block and N-version programming. In the latter a number ($N \geq 2$) of independently coded programs for a given function are run simultaneously (or nearly so) on loosely coupled computers, the results are compared, and in case of disagreement a preferred result is identified by majority vote (for $N>2$) or a predetermined strategy. This approach has been suggested in a general way by Elmendorf (11) and has more recently been developed into a practicable form by Avizienis and Chen (12,13) who report results on the use of this technique on a classroom problem.

The success of this technique is obviously governed by the degree of independence that can be achieved in the N versions of the program. The last reference states "Wherever possible, different algorithms and programming languages or translators are used in each effort." One might also want to see different data structures so that the common starting point for the N versions becomes the requirements document rather than the specification (which usually defines data structures quite rigorously). In addition, the voting algorithm and the housekeeping for 'results' prior to and after voting have been identified as critical items for this technique.

A specific constraint on N-version programming is the requirement for N computers that should be hardware independent yet able to communicate very efficiently so that rapid comparisons of results can be achieved. These

N computers must all be operating at the same time, and a hardware failure in any one of them will at best force the system into a different operating mode and may in minimal configurations cause loss of the fault-tolerance provisions. An example of a system that seems well suited to host N-version programming is SIFT (9). N-version programming is capable of masking intermittent hardware faults, and this can be an advantage in some applications. It also has the ability to aid in detection of permanent hardware faults, although detail fault-tolerance provisions for these may best be handled by dedicated hardware/software reconfiguration provisions.

The recovery block technique (14,15) can be applied to a more general spectrum of computer configurations, including a single computer (which may also include hardware fault tolerance). The simplest structure of the recovery block is

Ensure T

By P

Else by Q

Else Error

where T is the acceptance test condition that is expected to be met by successful execution of either the primary routine P or the alternate routine Q. The internal control structure of the recovery block will transfer to Q when the test conditions are not met by executing P. Techniques have been described for purging data altered during processing by P when Q is called (15).

For real-time applications it is necessary that the execution of a program be both correct and on time. For this reason the acceptance test is augmented by a watchdog timer that monitors that an acceptable result is furnished within a specified period. The timer can be implemented in either hardware or software or a combination of these. The structure of a recovery block for real time application modules is shown in Figure B-1 (16). In normal operation only the left part of the figure is traversed. When the acceptance test fails, or if the time expires, a transfer to the alternate call is initiated, a flag is set, and process Q is executed. If its result satisfies the acceptance test, the normal return exit in the right part of the figure is taken, and processing continues. If the acceptance test fails again, or if a time-out is encountered in the execution of Q (with the flag now set), an error return results.

A fault-tolerant navigation module using these techniques is described in the last reference, and the interaction of fault-tolerant application modules with the executive is also discussed there. Principles of a fault-tolerant scheduler based on the recovery block technique have also been described (17). The number of alternate routines is not restricted, and where it seems desirable any number of back-ups can be entered successively on failure of the acceptance test. Recovery blocks can also be used in concurrent processes and multi-level structures (18,19).

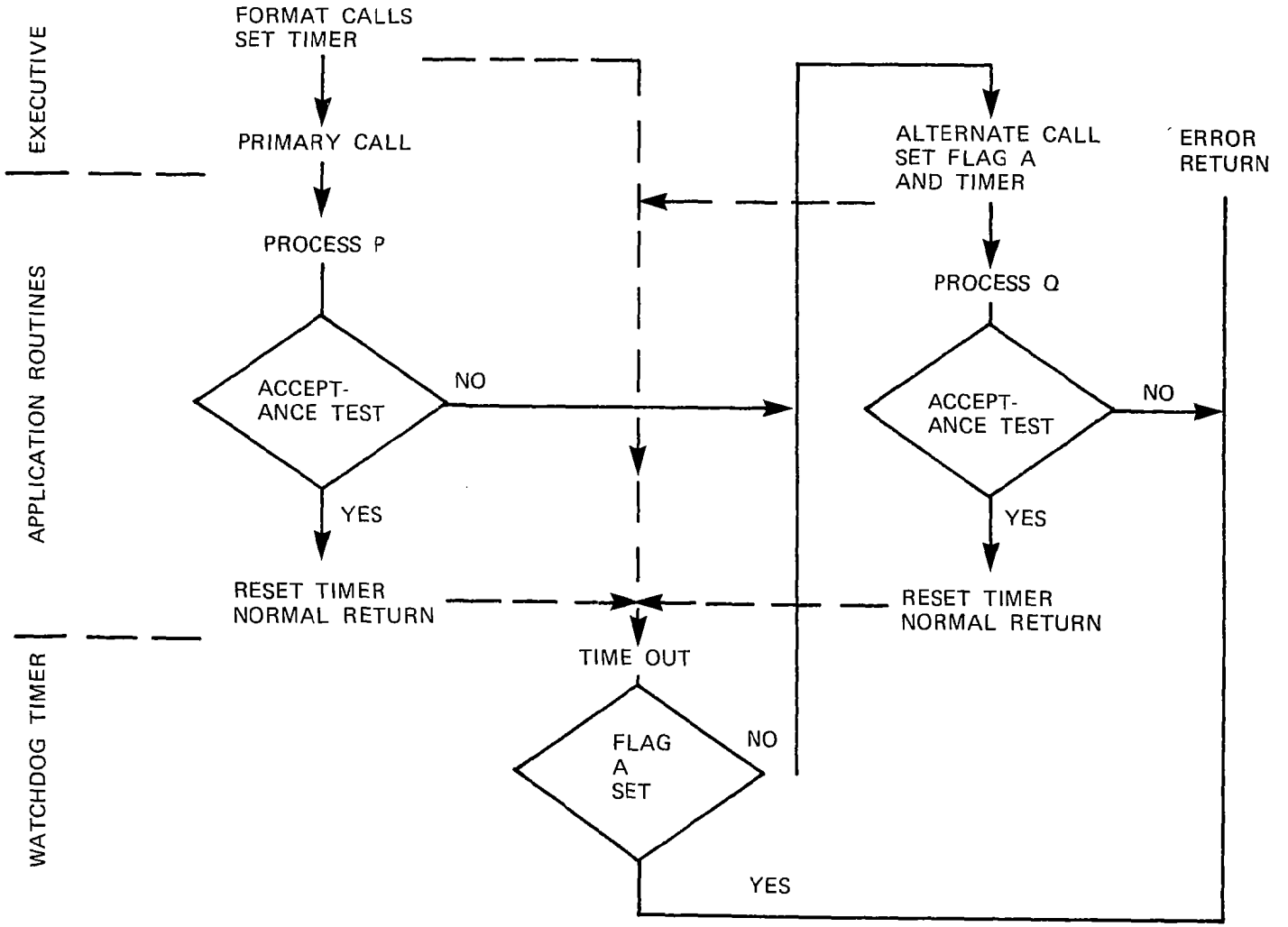

EXECUTIVE
APPLICATION ROUTINES
WATCHDOG TIMER
FORMAT CALLS
SET TIMER
PRIMARY CALL
PROCESS P
ACCEPT-
ANCE TEST
NO
YES
RESET TIMER
NORMAL RETURN
ALTERNATE CALL
SET FLAG A
AND TIMER
PROCESS Q
ACCEPT-
ANCE TEST
NO
YES
RESET TIMER
NORMAL RETURN
ERROR
RETURN
TIME OUT
FLAG
A
SET
NO
YES

Figure B-1.

As in N-version programming, it is desirable that the redundant routines in a recovery block be as independent of one another as possible. A specific and critical feature of the recovery block is the acceptance test. Alternate routines will be useless if failure of the primary one is not promptly detected. Thus acceptance tests should certainly be thorough. On the other hand, the acceptance test is traversed on every program execution, and the amount of code required for it should therefore be minimized. Few formal guidelines exist for satisfying these partly contradictory requirements. The following section attempts to classify techniques that have been used as a first step toward a systematic study of the design of acceptance tests.

ACCEPTANCE TESTS

Acceptance tests can be devised against two criteria: to detect deviations from expected program execution, or to prevent unsafe output. The first of these is more restrictive, and will result in more frequent transfer to the alternate routine. However, the penalties for unnecessary transfer are usually small, whereas the penalties for failure to switch when necessary can be much greater. For software that has been in use a long time, testing for unsafe output may be preferable because it can be simpler and it avoids unnecessary transfers. However, for programs just emerging from development, testing for expected program execution has some benefits:

- Unexpected behavior of the primary system will be noted even in cases where only a mild degradation is encountered. This aids in program evaluation.
- Switching to the alternate program is exercised more often under realistic (unplanned) conditions. Providing realistic testing of the fault-tolerance mechanism is a difficult undertaking.
- As a program matures it is usually easier to relax acceptance conditions than to make them more restrictive.

The four types of acceptance tests described below can usually be designed to test either for expected execution or for unsafe output.

Satisfaction of Requirements

In many cases the problem statement imposes conditions which must be met at the completion of program execution. These conditions can be used to construct the acceptance test.

In the 'Eight Queens' problem it is required that eight queens be located on a chessboard such that no two queens threaten each other. A suitable acceptance test for a computer program solving this problem is that the horizontal, vertical, and the two diagonals identified with the location of a given queen do not contain the location of any other queen. If testing for these conditions is already included in one or more of the

routines in a recovery block, then the acceptance test should use a different sequence and a different program structure.

The acceptance test for a sort problem described by Randell is also a test for satisfaction of requirements (15). The test involves checking at the completion of the execution that the elements are in uniformly descending order, and that the number of elements in the sorted set is equal to the number of elements of the original set. It is recognized that this test is not exhaustive: changes in an element during execution would not be detected. A stronger test, to determine that the elements of the sorted set are a permutation of the original set, was rejected because of excessive programming complexity and execution time.

An important subset in this class is the inversion of mathematical operations, particularly those for which the inverse is simpler than the forward operation. A typical example is the square root which is frequently handled as a subroutine call whereas squaring a number is a one-line statement. The effectiveness of inversion for the construction of acceptance tests is limited by the fact that some logical and algebraic operations do not yield a unique inverse, e.g., OR, AND, absolute value and trigonometric operations.

Testing for satisfaction of requirements is usually most effective when carried out on small segments of a computer program because at this level requirements can be stated in a simpler manner. On the other hand, for efficiency of the overall fault-tolerant software system, it is desirable to construct acceptance tests that cover large program segments. The classes of acceptance tests described in the following two sections have better capabilities in this regard but are more limited in the types of programs which they can handle. For text editing systems, compilers, and similar programs, tests for satisfaction of requirements constitute at present the most promising approach.

Accounting Checks

Commercial accounting had to struggle with the problem of maintaining accuracy in systems with many records and arithmetic operations long before the advent of the digital computer. Most of the procedures that had evolved for checking manual operations were taken over when bookkeeping evolved into data processing. These accounting checks can be very useful for acceptance tests in software that serves transaction-oriented applications. Airline reservation systems, library records, and the dispensing of dangerous drugs all can be checked by these procedures.

The most rudimentary accounting check is the tally which in computer usage has become the checksum. Whenever a volume of financial records (checks, invoices, etc.) is transmitted among processing stations, it is customary to append a tally slip representing the total amount in the records. On receipt, a new total is computed and compared with the tally. The corresponding use of checksums is widespread in data processing. The digital presentation of information makes it possible to apply the checksum to non-numerical information as well.

When a large volume of records representing individual transactions is aggregated, it is almost impossible to avoid errors to incorrect transcriptions or due to lost or misrouted documents. In the commercial environment such errors are not always of an innocent nature since an employee may be able to pocket the amount corresponding to an improper entry. The double entry bookkeeping system evolved as an effective means of detecting such errors. In this procedure the total credits for all accounts must equal the total debits for all accounts for any arbitrary time period, provided only that corresponding transactions have been entered into all accounts. This equality can also be used as a criterion in acceptance tests.

Another accounting check that may be of use as an acceptance test is the reconciliation of authorized transactions with changes in physical inventory over a period of time. For example, in storage of nuclear material it is possible to determine the quantity in inventory by means of radiation counters which can feed data directly into a computer. At specified intervals the change in radiation level can be compared to that independently calculated for normal decay of the material and authorized inventory transactions. This furnishes an overall check, including most computer and software errors.

Accounting checks are suitable only for transaction-oriented applications and they cover only elementary mathematical operations. Within this sphere, accounting checks provide a time-tested and demonstrably complete means for assessing the correctness of computer operations. They can test recovery blocks containing large software segments, and portions of the input operations and physical inventory can be checked in the same process.

Reasonableness Tests

This heading includes acceptance tests based on precomputed ranges of variables, on expected sequences of program states, or on other relationships that are expected to prevail for the controlled system. The dividing line between reasonableness tests and testing for satisfaction of requirements can become blurred, but in general reasonableness tests are based on physical constraints whereas testing for requirements uses primarily logical or mathematical relationships.

Reasonableness tests for numerical variables may examine the individual value (e.g., to be within range), increments in individual values of the same variable (increments between successive values or deviations from a moving average), or the correlation between values of different variables or of their increments. Examples of these different types are examined for an airspeed calculation. The indicated airspeed (typically an input quantity), and the true airspeed (a computed quantity) must each be within a range that is dictated by the aerodynamic and structural capabilities of the airframe, e.g., 140 to 1100 km/hr. Obviously only gross malfunctions of either the sensor or of the computing process can be diagonosed by such a test. The airspeed range is a function of aircraft configuration (flap position, etc.) and an acceptance test with narrower limits can be constructed if adjustments for configuration are included.

A much more sensitive test for sudden malfunctions (and these are usually the most critical ones) can be devised by examining the increments in each quantity. Changes in speed are equivalent to acceleration, and in the normal flight mode changes in forward speed are well below the 1 g level (9.81 m/sec/sec = 35.3 km/hr/sec). Even if the acceptance test is based on the maximum allowable acceleration for structural integrity (which may be 6 g), the corresponding change in speed is limited to 213 km/hr/sec. Individual speed calculations may be carried out ten times per second, yielding an allowable speed increment between successive values of 21.3 km/hr. To suppress noise in the sensor output, the acceptance test may operate on averaged readings so that the effective sampling interval is increased, but, even under these circumstances, the acceptance test based on increments will for sudden malfunctions be much more sensitive than one based on range.

The correlation between increments of indicated and true airspeed can be used for further refinement, but an even more useful correlation can be obtained by comparing increments in true airspeed with the acceleration measured by an appropriately oriented accelerometer. In this case, limits of the acceptance test will depend on the noise characteristics of the instruments used, elastic deformations of the aircraft, and other secondary characteristics. For filtered observations the acceptance region can probably be reduced by an order of magnitude over that obtained in the test based on increments alone.

The use of correlated measures for acceptance tests always raises the problem that errors might be introduced by the variable added to provide the refinement (in this case the accelerometer output). The consequences of a spurious failure of the acceptance test must be evaluated to determine whether the refinement is indeed warranted. To be particularly avoided is the use of a variable in the acceptance test that is also used in the back-up routine because this could cause transfer to the back-up at exactly the time when its data source is unreliable. The importance of keeping the back-up program independent of software structures and data used in the acceptance test is discussed further in connection with reliability modeling.

An example of a reasonableness test based on state transition can be found in an electronic telephone switching system. Once a call has proceeded to the 'connected' state it is inadmissible for it to subsequently go to 'ringing' or 'busy.' Inappropriate transitions of this type can therefore be used to signal the need for switching to an alternate routine.

Tests for reasonableness of numerical or state variables are a very flexible and effective way of constructing acceptance tests for fault-tolerant software. They permit acceptance criteria to be modified as a program matures. Reasonableness tests can be devised for most real-time programs that control physical variables, and they may monitor overall performance of a computing system, e.g., by reasonableness tests on output variables.

Computer Run Time Checks

Most current computers provide continuous hardware-implemented testing for anomalous states such as divide by zero, overflow and underflow, attempts to execute undefined operation codes, or writing into write-protected memory areas. If such a condition is detected, a bit in a status register is set, and subsequent action can then be defined by the user. When fault-tolerant software is being executed, encountering one of these conditions can be equated with failure of the acceptance test and transfer to an alternate software routine is then effected. The previously mentioned watchdog timer can also be tied into this status reporting scheme.

Run time checks can also incorporate data structure and procedure oriented tests that are imbedded in special support software or in the operating system. Checking that array subscripts are within range is already implemented in many current computer systems. Array value checking (for being within a given range, being in ascending or descending order, etc.) has also been proposed (20). Under the title "Self-checking Software" and "Error-Resistant Software" a number of interesting run-time monitoring techniques have been described, many of which are akin to acceptance tests mentioned earlier in this section (21,22). A particularly appropriate concept for a run-time acceptance test mentioned in the last reference is an Interaction Supervisor. In its simplest form this requires declaration for each module of authorized callers and authorized calls. The Interaction Supervisor will cause failure of the acceptance test if access to, or exit from, a module involves unauthorized locations.

The value of run time checks is not restricted to prevention of failures due to errors arising directly from the attribute that is being monitored. They cover a much wider area, e.g., attempts to write into write-protected memory may have as their original cause an improper indexing algorithm, a failure to clear a register, or similar more subtle software discrepancies. It is therefore appropriate to use all of these facilities that modern computers, operating systems, and programming languages can contribute for the implementation of acceptance tests even if the occurrence of the monitored conditions per se could be prevented by other means. Run time checks are not exhaustive but they also require very little development time or other resources. They supplement the previously mentioned types of acceptance tests for critical segments, and they can be used by themselves as the acceptance test for non-critical segments operating as part of an overall fault-tolerant software system.

Ultimately one would like to see a classification of acceptance tests that characterize them by error detection capability, run time penalty, and storage requirements, thus permitting a rational selection for each application. This stage, unfortunately, has not yet been reached. But that need not deter advancing with practical applications; there is little methodological basis for the routine testing of software which is being carried out all the time, and occasionally even with satisfactory results.

SYSTEM RELIABILITY MODELING

A potential advantage of the fault-tolerant software approach over other techniques for software reliability improvement is that it permits the employment (with some modifications) of system reliability modeling techniques that have been developed in the hardware field. The qualifying 'potential' in the previous sentence reflects the fact that adequate parameters for the models do not yet exist although a methodology for obtaining them has been defined (23,24). Interesting insights can be achieved by modeling even at the present stage.

This will be demonstrated with a transition model shown in Figure B-2 for a recovery block consisting of a primary and an alternate routine (16). Reliability models can also be developed for N-version programming and for recovery blocks having multiple alternates.

Along the bottom of the figure are shown four possible states for the recovery block (here identified as an application module). Starting at state 1, primary routine operating, the immediate transitions are

Loop 1,1 - Primary routine continues to operate

Arc 1,2 - Failure in primary routine detected

Arc 1,4 - Undetected failure in the primary routine

Out of state 2 a transient state after detection of failure, two transitions are possible:

Arc 2,3 - Transition to a satisfactory alternate

Arc 2,4 - Failure at transition

State 3 is defined as satisfactory operation of the back-up routine through at least one complete program pass, and once this is achieved, the further possible transitions are

Loop 3,3 - Back-up continues to operate

Arc 3,1 - Reversion to primary routine

Arc 3,4 - Uncorrelated failure of the back-up

The word "uncorrelated" is inserted because failures in the back-up routine occasioned by the transition from state 2 are represented by the arc 2,4 as will be further discussed. The uncorrelated failures (arc 3,4) are extremely unlikely, given thoroughly tested software and the usually limited time period for operation in the back-up mode. This transition is therefore shown in dashed symbols.

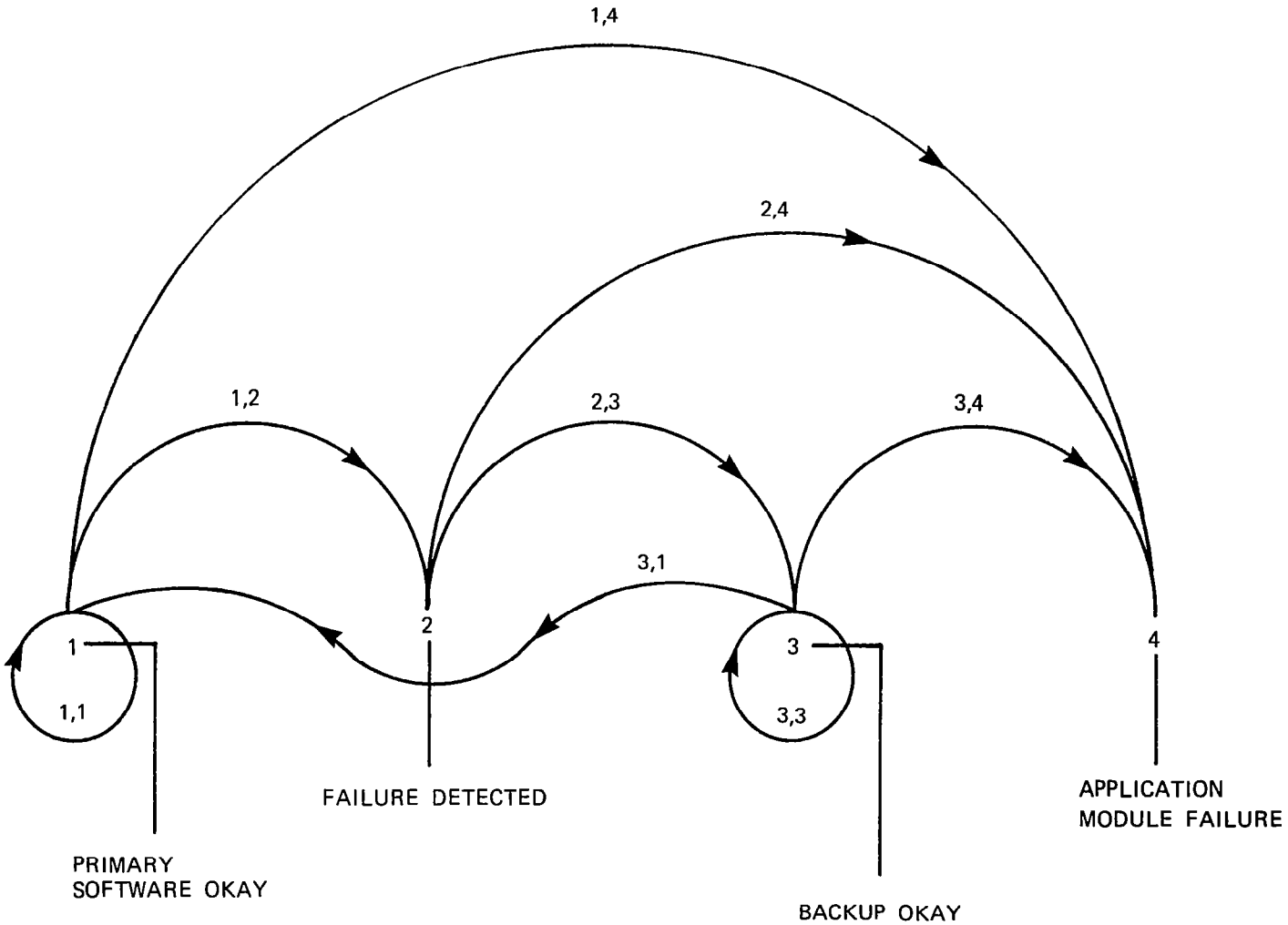

1,4
2,4
1,2
2,3
3,4
3,1
1
1,1
2
3
3,3
4
PRIMARY
SOFTWARE OKAY
FAILURE DETECTED
BACKUP OKAY
APPLICATION
MODULE FAILURE

Figure B-2.

The major sources for failure of the recovery block are expected to be undetected failures of the primary routine and correlated failures in the back-up. The former are caused by deficiencies in the acceptance test. The previous discussion of this subject has shown that design of acceptance tests is far from an established discipline, and the possibility of undetected failures must be accounted for. Correlated failure of the back-up routine (arc 2,4) can be caused by two circumstances: correlated deficiencies in the primary and back-up routine software, and correlation of faults in the acceptance test with those in the back-up routine. Insistence on independent design and coding of the two software routines, and, wherever possible, use of independent data sources, will minimize the first of these. The second cause of these correlated failures is a particularly insidious one, since the failure occurs even though the primary software may execute correctly! Possible sources of such correlated failures of the acceptance test and the back-up routine must therefore be thoroughly investigated in the design of a recovery block. The use of common data, common algorithms, and common subroutines should be avoided. Where some commonality is unavoidable (cf. the Eight Queens problem in section 3) at least the detailed software design should be varied.

Use of this transition model, even with the very inadequate data currently available, has led to some interesting insights into the measures necessary to obtain software reliability consistent with critical flight control applications (17). A substantial advantage of the recovery block approach illustrated by this model is that the requirement for demonstrated reliability of the primary and alternate routines can be held several orders of magnitude (in terms of failure probability) below that required for the recovery block as a whole.

DIRECTIONS FOR THE FUTURE

Complete fault-tolerant software systems can for the foreseeable future be considered only for the most demanding and safety-critical applications, e.g., fly-by-wire passenger aircraft or safety systems for nuclear reactors. But fault-tolerant segments in otherwise conventional software may find much wider use. Such segments may be needed only temporarily, e.g., when a new operating system is being introduced, or they may be permanently installed, e.g., for back-up file management in a reservation or inventory control system.

Even more widespread may be the use of fault-tolerant techniques (short of a formal recovery block or N-version segment). In many applications it may be sufficient to halt operations (or to flag output) when errors occur. Acceptance tests or the previously referenced techniques for self-checking software will be used here. In other tasks, e.g., in the accounting field, the availability of back-up routines and data caches may be important, while the acceptance test might be relegated to a human observer or to a separately running audit routine.

Research will become more directed to the application of the basic techniques outlined here to multi-tasking and multi-processing environments, and to the multilayered operating systems of time-shared computers.

All of the present and most of the foreseeable applications are dictated by a need for greater reliability in the computing function without specific economic trade-offs of one technique of achieving this versus another (the choices are very limited). It is questionable whether at some future time good criteria can be developed on where to apply software fault-tolerance and where to apply intensive validation methods. Where highly reliable illumination is desired we employ long life bulbs, redundant bulbs, and separate emergency lighting systems. Sometimes all of these are employed together, and sometimes separately. Any one of them is better than reliance on a single standard light bulb. Future generations may look in the same way on our efforts to produce more reliable software.

ACKNOWLEDGEMENT

Portions of the work reported here were carried out under subcontracts to C. S. Draper Laboratory and SRI International in connection with efforts by these organizations for the NASA Langley Research Center under contracts NAS1-15336 and NAS1-15428, respectively.

REFERENCES

1. Proceedings of the IEEE, Special Issue on Fault-Tolerant Digital Systems, Vol. 66 No. 10, Oct. 1978.

2. W. E. Howden, "Theoretical and Empirical Studies of Software Testing," IEEE Transactions on Software Engineering Vol. SE-4 No. 4, July 1978, pp. 293-297.

3. S. L. Gerhart and L. Yelowitz, "Observations on the Fallibility in Applications of Modern Programming Methodologies," IEEE Transactions on Software Engineering, Vol. SE-2, No. 3., Sept. 76, pp. 195 - 207.

4. C. Reynolds and R. T. Yeh, "Induction as the Basis for Program Verification," IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, Dec. 1976, pp. 244 - 252.

5. S. L. Gerhart, "Program Verification in the 1980s: Problems, Perspectives, and Opportunities," ISI/RR-78-71, Information Sciences Institute, Marina del Rey, Calif., August 1978.

6. L. A. Belady and M. M. Lehman, "A Model of Large Program Development," IBM Systems Journal, Vol. 15, No. 3, (1976) pp. 225 - 252.

7. W. N. Toy, "Fault Tolerant Design of Local ESS Processors," in (1) pp. 1126 - 1145.

8. A. L. Hopkins et al., "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft," in (1) pp. 1221 - 1239.

9. J. H. Wensley et al., "SIFT: The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control, in (1) pp. 1240 - 1254.

10. K. H. Kim et al., "Strategies for Structured and Fault-Tolerant Design of Recovery Programs," Proceedings of COMPSAC '78, pp. 651 - 656 (Nov. 78).

11. W. R. Elmendorft, "Fault-Tolerant Programming," Digest of the 1972 International Symposium on Fault-Tolerant Computing, pp. 79 - 83.

12. A Avizienis and L. Chen, "On the Implementation of N-Version Programming for Software Fault-Tolerance During Execution," Proceedings of COMPSAC '77, pp. 149 - 155 (Nov. 77).

13. L. Chen and A. Avizienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," Digest of Papers - FTCS - 8, pp. 3 - 9 (June 1978)

14. J. J. Horning et al., "A Program Structure for Error Detection and Recovery," Proceedings of the Conference on Operating Systems: Theoretical and Practical Aspects, IRIA, April 1974, pp. 174 - 193.

15. B. Randell, "System Structure for Software Fault-Tolerance," IEEE Transactions on Fault-Tolerant Software, Vol. SE-1, No. 2, June 1975, pp. 220 - 232.

16. H. Hecht, "Fault-Tolerant Software for Real-Time Applications," ACM Computing Surveys, Vol. 8, No. 4, Dec. 1976, pp. 391 - 407.

17. Advanced Programs Division, The Aerospace Corporation, "Fault-Tolerant Software Study," NASA CR 145298, February 1978.

18. J. S. M. Verhofstad, "The Construction of Recoverable Multi-Level Systems," Ph.D. Dissertation, University of Newcastle upon Tyne, August 1977.

19. K. H. Kim and C. V. Ramamoorthy, "Failure-Tolerant Parallel Programming and its Supporting System Architectures," AFIPS - Conference Proceedings, Vol. 45 (NCC 1976) pp. 413 - 423.

20. L. G. Stucki and G. L. Foshee, "New Assertion Concepts for Self-Metric Software Validation," Proc. of the 1975 International Conference on Reliable Software, IEEE Cat. 75CH0940-7CSR, pp. 59 - 71, April 1975.

21. S. S. Yau and R. C. Cheung, "Design of Self-Checking Software," Proc. of the 1975 International Conference on Reliable Software, IEEE Cat. No. 75 CHO 940-7CSR, pp. 450 - 457, April 1975.

22. S. S. Yau, R. C. Cheung and D. C. Cochrane, "An Approach to Error-Resistant Software Design," Proc. of the Second International Conference on Software Engineering, IEEE Cat. No. 76CH1125-4C, pp. 429 - 436, October 1976.

23. J. D. Musa, "Measuring Software Reliability," ORSA/TIMS Journal, May 1977, pp. 1 - 25.

24. H. Hecht et al., "Reliability Measurement During Software Development," NASA CR-145205, September 1977.

APPENDIX C - PARTICIPANTS

Mr. S. J. Bavuso
NASA, Langley Research Center
Mail Stop 477
Hampton, VA 23665
Tel: (804) 827-3681

Ms. Lynn Buhler
ORI, Inc.
1400 Spring Street
Silver Spring, MD 20910
Tel: (301) 588-6180

Dr. Wm. C. Carter
IBM Research,
T. J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
Tel: (914) 945-1092

Mr. R. L. Coffin
Rockwell International
400 Collins Road, N.E.
Cedar Rapids, IA 52406
Tel: (319) 395-3057

Dr. Donald D. Dillehunt
Rockwell International
Mail Stop FB-37
12214 Lakewood Blvd.
Downey, CA 90241
Tel: (213) 922-4157

Mr. Billy Dove
NASA, Langley Research Center
Mail Stop 477
Hampton, VA 23665
Tel: (804) 827-3681

Mr. G. Robert Flynn
The Boeing Co.
P.O. Box 3707 Mail Stop 3N-43
Seattle, WA 98124
Tel: (206) 773-1786

Dr. Susan Gerhart
USC/Information Sciences Institute
4676 Admiralty Way
Marina Del Rey, CA 90291
Tel: (213) 822-1511

Mr. Jack Goldberg
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025
Tel: (415) 326-6200 Ext. 2784

Mr. John E. Hart
Lockheed-Georgia Co.
Dept. 72-02, Zone 13
Marietta, GA 30063
Tel: (404) 424-4789

Dr. Herb Hecht
SoHaR Inc.
1040 S. La Jolla Ave.
Los Angeles, CA 90035
Tel: (213) 935-7039

Mr. Dale G. Holden
NASA, Langley Research Center
Mail Stop 477
Hampton, VA 23665
Tel: (804) 827-3681

Dr. Albert L. Hopkins, Jr.
C.S. Draper Laboratory Inc.
Mail Station 35
555 Technology Square
Cambridge, MA 02139
Tel: (617) 258-1451

Mr. Al Lindler
NASA, Langley Research Center
Mail Stop 477
Hampton, VA 23665
Tel: (804) 827-3681

Mr. Brian Lupton
NASA, Langley Research Center
Mail Stop 477
Hampton, VA 23665
Tel: (804) 827-3681

Mr. Wm. R. Mann
Rockwell International
Mail Stop 503-240
4311 Jamboree Road
Newport Beach, CA 92660
Tel: (714) 833-4013

Dr. Gerald M. Masson
Electrical Engineering Dept.
Johns Hopkins Univ.
Barton Hall
Charles & 34th Sts.
Baltimore, MD 21218
Tel: (301) 338-7013

Prof. Edward J. McCluskey
Stanford University
Computer Systems Lab.
Stanford, CA 94305
Tel: (415) 497-1451

Mr. P. M. Melliar-Smith
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025
Tel: (415) 326-6200 Ext. 2336

Dr. John F. Meyer
2523 E. Engineering Bldg.
The Univ. of Michigan
Ann Arbor, MI 48109
Tel: (313) 763-0037

Mr. Earle Migneault
NASA, Langley Research Center
Mail Stop 477
Hampton, VA 23665
Tel: (804) 827-3681

Mr. Kurt Moses
Bendix Corp.
Flight Systems Div., Dept. 7211
Teterboro, NJ 07608
Tel: (201) 288-2000 Ext. 1584

Mr. Nicholas D. Murray
NASA, Langley Research Center
Mail Stop 477
Hampton, VA 23665
Tel: (804) 827-3681

Dr. John M. Myers
77 N. Washington Street
Boston, MA 02114
Tel: (617) 277-5301

Mr. Bill Patten
ORI, Inc.
1400 Spring Street
Silver Spring, MD 20910
Tel: (301) 588-6180

Mr. Wm. C. Rogers
ORI, Inc.
1400 Spring Street
Silver Spring, MD 20910
Tel: (301) 588-6180

Mr. C. L. Seacord
Honeywell, Inc.
Mail Station MN 15-2644
1625 Zarthan Ave.
St. Louis Park, MN 55416
Tel: (612) 542-5758

Dr. Richard K. Smyth
Milco International Inc.
15628 Graham Street
Huntington Beach, CA 92649
Tel: (714) 894-1717

Mr. Larry Spencer
NASA, Langley Research Center
Mail Stop 477
Hampton, VA 23665
Tel: (804) 827-3681

Dr. J. J. Stiffler
Raytheon Co.
Box 3190
528 Boston Post Road
Sudbury, MA 01776
Tel: (617) 443-9521 Ext. 2771

Mr. Steven Toth
ORI, Inc.
1400 Spring Street
Silver Spring, MD 20910
Tel: (301) 588-6180

Mr. John H. Wensley
August Systems Inc.
1940 McGilchrist Street, S.E.
Salem, OR 97302
Tel: (503) 364-1000

Mr. Allan White
Kentronics
3221 N. Armistead
Hampton, VA 23665

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CP-2114		

4. Title and Subtitle	5. Report Date
VALIDATION METHODS FOR FAULT-TOLERANT AVIONICS AND CONTROL SYSTEMS - WORKING GROUP MEETING I	December 1979
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
ORI, Incorporated, Compilers	L-13436
	10. Work Unit No.
9. Performing Organization Name and Address	
NASA Langley Research Center Hampton, VA 23665	11. Contract or Grant No.
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address	Conference Publication
National Aeronautics and Space Administration Washington, DC 20546	14. Sponsoring Agency Code

15. Supplementary Notes

ORI, Incorporated: Silver Spring, Maryland

16. Abstract

This document presents the proceedings of the First Working Group Meeting on validation methods for fault-tolerant computer design, held at Langley Research Center, March 12-14, 1979. The first Working Group meeting provided a forum for the exchange of ideas. The state of the art in fault-tolerant computer validation was examined in order to provide a framework for future discussions concerning research issues for the validation of fault-tolerant avionics and flight control systems. The activities of the Working Group were structured during the two-day session to focus the participants' attention on the development of positions concerning critical aspects of the validation process. These positions are presented in this report in three sections. In order to arrive at these positions, the Working Group discussed the present state of the art in five sessions:

1. Fault-Tolerant Avionics Validation
2. Program (Software) Validation
3. Device Validation
4. Safety, Reliability, Economics and Pérformance Analysis
5. Fault Models

17. Key Words (Suggested by Author(s))		18. Distribution Statement
Avionic systems	Testing	Unclassified - Unlimited
Validation process	Software reliability	
Design proof	Reliability models	
Systems reliability		
		Subject Category 59

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price*
Unclassified	Unclassified	110	$6.50

* For sale by the National Technical Information Service, Springfield, Virginia 22161

NASA-Langley, 1979